Helmut E. Graeb
Editor

Analog Layout Synthesis

A Survey of Topological Approaches

Springer

VISIT…

Analog Layout Synthesis

Helmut E. Graeb
Editor

Analog Layout Synthesis

A Survey of Topological Approaches

Springer

Editor
Helmut E. Graeb
Technische Universität München
Munich
Germany
graeb@tum.de

ISBN 978-1-4419-6931-6 e-ISBN 978-1-4419-6932-3
DOI 10.1007/978-1-4419-6932-3
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2010935721

© Springer Science+Business Media, LLC 2011
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Analog components appear on 75% of all chips, and cause 40% of the design effort and 50% of the design errors detected after first silicon measurements, reported *EDA Weekly* on March 21, 2005. Due to increasing functional complexity of system-on-chips, the difficulties in analog design and the lack of design automation support for analog circuits continually increase the bottleneck character of analog components in chip design. Design methodology and design automation for analog circuits therefore is a crucial problem for future system-on-chips.

Eminently critical is the layout synthesis part of the analog design flow. Although there have been a lot of very good works from universities over the years, some of which even found their way to commercial EDA tools, industrial application of analog layout synthesis is still in its infancy when it is compared to its digital counterpart! The industrial point of view even says that practicable EDA tools for analog layout synthesis did not exist.

But it seems that this situation is about to change. In the face of increasing circuit complexity and high performance SoC designs, the once-sleepy analog EDA market is experiencing an increasing shift from single vendor solutions to design tool integration via alliances between many players. The attempt to create an inter-platform reference, such as the Interoperable PDK Libraries (IPL) alliance, where analog layouts made with a tool can be imported error-free to different frameworks, is an example. Many EDA start-ups as well as major leaders are already announcing key automated layout tools for the analog designer intended to boost his/her productivity.

In this exciting scenario, academia continues to strive for new, more efficient, and complementary approaches to this task and to the existing tools, and has recently produced some very interesting new solutions. The intention of this book has two parts. On the one hand, it summarizes and presents these latest results. On the other hand, it is dedicated to give an introduction to advanced analog layout methods on the graduate level.

The book is structured in three parts. The first part with three chapters covers recent approaches to topological placement of analog circuits. The second part treats the problem of routing. The third part with three more chapters deals with layout in the design flow, namely, with the problem of retargeting an existing layout for a new technology, with integrating layout in the sizing process, and with constraint management in the design flow.

The first chapter starts with an introduction to the different ways of approaching in CAD tools device-level placement problems for analog layout. It is elaborated how the structural representation of the layout in the algorithm is crucial for the efficiency and efficacy of the placement process. Besides the classical way of using absolute coordinates for the module placement and slicing structures for topological representations, which encode the relative positioning between cells, it describes how the sequence-pair and tree-based topological representation can be applied to dramatically reduce the search space to the tiny fraction, which satisfies the inherent symmetry constraints in analog circuits. It further develops sufficient conditions to ensure the symmetry constraints during the successive moves of a placement algorithm and, based on these ideas, presents several topological algorithms that perform the exploration process very efficiently.

The second chapter furthers the ideas presented in the first chapter and extends them to a hierarchical module clustering. The analog devices can be hierarchically clustered into groups according to models, circuit functionalities, or signal/current flows. Following the B*-tree, a hierarchical B*-tree (HB*-tree) placement representation is developed to model this circuit hierarchy and symmetry and proximity constraints among modules and across the hierarchy. This hierarchical representation is fed into a placement algorithm to generate optimum device placements that meet all device layout constraints. Performing a simulated annealing algorithm, the placement of the device modules in different device groups belonging to different clustering hierarchies is simultaneously optimized.

The third chapter first introduces a method to automatically derive the circuit hierarchy and the resulting symmetry, proximity, and matching constraints from a netlist. A deterministic algorithm is then presented that computes the shape function of different aspect ratios of the circuit placement by a recursive bottom-up approach through the derived circuit hierarchy starting from basic modules such as current mirrors or differential pairs. For each hierarchy level, the shape function is determined by combining the placements of the next-lower hierarchy. These are stored as so-called enhanced shape functions that include the corresponding B*-trees of each individual shape. Algorithms are proposed to generate the vertical and horizontal sum of two B*-Trees of placements while provably complying with the constraints. As the algorithm bounds the enumeration according to the circuit hierarchy and the constraints, it generates results very fast, while being deterministic without any tuning parameter.

The second part of the book deals with analog routing. It gives a tutorial on routing methods and corresponding placement and routing representations, including constraints, for instance, for symmetry or crosstalk. A review of different routing strategies and the corresponding state of the art follows. Early routing approaches inspired from digital design, cost-driven approaches, and parasitic-driven approaches (including, e.g., performance sensitivities), as well as the A* algorithm are covered. The connection to placement through templates and other integration approaches is discussed afterward. Then, the partitioning of routing into global and detailed routing, as in digital design, is described. The chapter concludes with specialized routing approaches for RF circuits and analog arrays.

The third part of the book addresses analog layout issues arising from the ambient design flow.

In Chap. 5, the task of retargeting an existing layout, including placement and routing, is examined. Specific algorithms for layout retargeting may be beneficial if the involved layout modifications are moderate or to extract and conserve the knowledge contained in a layout. After a short introduction to the preparatory steps of layer mapping, constraint generation and device recognition, the main algorithmic step of retargeting, i.e., layout compaction, is described in detail. Based on the linear programming approach to its solution, a graph-based simplex method is presented with full details. The different types of constraints, the complexity of the algorithm, and practical issues are discussed as well.

Chapter 6 is dedicated to the problem of integrating layout effects into the circuit sizing process, to avoid unnecessary iterations between electrical and physical synthesis as much as possible. This has been called parasitic-aware synthesis. This chapter reaches from the very basics (what is it, and why and when is it really necessary) to a practical implementation of this type of synthesis process. Different methods to carry it out as well as their pros, cons, and trade-offs (mainly efficiency vs. completion time) will be explained. A technique will be presented that uses a combination of simulation-based optimization, procedural layout generation, exhaustive geometric evaluation algorithms, and several mechanisms for parasitic estimation, to comprehensively incorporate the layout-induced parasitic into electrical synthesis.

Chapter 7 concludes the book with a discussion of the management of the crucial factor in analog layout — the constraints. It provides a problem formulation for the classification, representation, transformation, and verification of constraints in a top-down design flow, as well as a formulation of a constraint engineering system, including its impact on the design flow and its algorithms.

This bow from placement to routing to the design flow, drawn by the structure of the book, invites the reader to start from the beginning and read one chapter after the other. At the same time, the chapters are self-contained and may be accessed individually and independently. In any way she or he approaches the book, the reader will gain a deep insight into the tasks of analog layout and into the actual solution approaches.

Munich *Helmut Graeb*
March 2010

The Authors

Hazem Abbas (*Hazem_Abbas@mentor.com*) received the B.Sc. and M.Sc. degrees in 1983 and 1988, respectively, from Ain Shams University, Egypt and the Ph.D. degree in 1993 from Queen's University at Kingston, Canada all in Electrical and Computer Engineering. He held a postdoc position at Queen's University in 1993. In 1995, he worked as Research Fellow at the Royal Military College at Kingston and then joined the IBM Toronto Lab as a Research Associate. He joined the Department of Electrical and Computer Engineering at Queen's University as an Adjunct Assistant Professor in 1997–1998. He is now with the Department of Computers and Systems Engineering at Ain Shams University, Egypt, as a Professor on Computer and Systems Engineering. Dr. Abbas is also working with Mentor Graphics Inc., Egypt as a Senior Engineering Manager. His research interests are in the areas of neural networks, pattern recognition, evolutionary computations, and image processing and their parallel and multicore implementations. He also serves as the President of the IEEE Signal Processing Chapter in Cairo.

Florin Balasa (*balasa@suu.edu*) received the Ph.D. degree in computer science from the Polytechnical University of Bucharest, Bucharest, Romania, in 1994, and the Ph.D. degree in electrical engineering from the Katholieke Universiteit Leuven, Leuven, Belgium, in 1995.

From 1990 to 1995, he was with the Interuniversity Microelectronics Center (IMEC), Leuven, Belgium. From 1995 to 2000, he was a Senior Design Automation Engineer at the Advanced Technology Division of Conexant Systems (formerly Rockwell Semiconductor Systems), Newport Beach, CA. He is currently an Associate Professor of Computer Science at the Southern Utah University.

Dr. Balasa was a recipient of the US National Science Foundation CAREER Award.

Rafael Castro-López (*castro@imse-cnm.csic.es*) received the "Licenciado en Física Electrónica" degree (M.S. degree on Electronic Physics) and the "Doctor en Ciencias Físicas" (Ph.D. degree) from the University of Seville, Spain, in 1998 and 2005, respectively. Since 1998, he has been working at the Institute of Microelectronics of Seville (CSIC-IMSE-CNM) of the Spanish Microelectronics Center, where he now holds the position of Tenured Scientist. His research interests lie in the field of integrated circuits, especially design and computer-aided design

for analog and mixed-signal circuits. He has participated in several national and international R&D projects and co-authored more than 50 international scientific publications, including journals, conference papers, book chapters, and the book *Reuse-based Methodologies and Tools in the Design of Analog and Mixed-Signal Integrated Circuits* (Springer, 2006).

Yao-Wen Chang (*ywchang@cc.ee.ntu.edu.tw*) received the B.S. degree from National Taiwan University in 1988, and the M.S. and Ph.D. degrees from the University of Texas at Austin in 1993 and 1996, respectively, all in computer science.

Currently, he is Professor of the Department of Electrical Engineering and the Graduate Institute of Electronics Engineering, National Taiwan University, Taipei, Taiwan. His current research interest lies in electronic design automation, with an emphasis on physical design for nanometer IC's and design for manufacturability. He has co-edited one textbook on Electronic Design Automation and co-authored one book on routing and more than 180 ACM/IEEE conference/journal papers in these areas. Dr. Chang is a four-time winner of the ACM ISPD contests (1st and 3rd places in the respective 2009 and 2010 Clock Network Synthesis Contests, second place in the 2008 Global Routing Contest, and third place in the 2006 Placement Contest). Dr. Chang received seven excellent teaching awards, Best Paper Awards at the 2007 and 2008 VLSI Design/CAD Symposia, a Best Paper Award at ICCD-95, and 14 Best Paper nominations from DAC (four times), ICCAD (three times), ISPD (four times), TODAES, ASP-DAC, and ICCD.

Dr. Chang is currently an associate editor of IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD) and an editor of the Journal of Information Science and Engineering (JISE). He has served as the ACM ISPD General Chair, and the ACM ISPD and IEEE FPT Technical Program Chairs, on the IEEE/ACM ICCAD Executive Committee, the IEEE/ACM ASP-DAC Steering Committee, the ACM/SIGDA Physical Design Technical Committee, the IEEE CEDA Conferences Committee, and the technical program committees of major EDA conferences, including ASP-DAC, DAC, DATE, FPL, FPT, GLSVLSI, ICCAD, ICCD, IECON, ISPD, SOCC, TENCON, and VLSI-DAT. He has served as the chair of the EDA Consortium of the Ministry of Education of Taiwan, Principal Reviewer of the SBIR Projects of the Ministry of Economic Affairs, Review and Planning Committee Member of the National Science Council of Taiwan, Independent Board Director of Genesys Logic, Inc., Technical Consultant of Faraday Technology Inc., MediaTek Inc., and RealTek Semiconductor Corp., and Member of Board of Governors of Taiwan IC Design Society.

Mohamed Dessouky (*Mohamed_Dessouky@mentor.com*) received the Ph.D. degree in electrical engineering from the University of Paris VI, France, in 2001. In 1992, he joined the Electronics and Electrical Communications engineering department, University of Ain Shams, Egypt, where he was a Research and Teaching Assistant, and is currently an Associate Professor. Since 2004, he is on leave to Mentor Graphics Inc., Egypt, where he served as the head of the analog design team

in the IP Division and is currently a Staff Engineer. His research interests include the design of switched-capacitor circuits, analog-to-digital conversion and CAD for analog, RF and mixed-signal integrated circuits.

Günhan Dündar (*dundar@boun.edu.tr*) got his B.S. and M.S. degrees from Boğaziçi University, Istanbul, Turkey in 1989 and 1991, respectively, and his Ph.D. degree from Rensselaer Polytechnic Institute in 1993, all in electrical engineering. Since 1994, he has been with the Department of Electrical and Electronic Engineering, Boğaziçi University, where he is currently a professor, with some temporary positions at the Turkish Naval Academy, EPFL (Lausanne, Switzerland), and the Technical University of Munich during this period.

Dr. Dündar has published more that 100 papers in international journals and conferences and a book on analog design automation. During his career, he has received various awards, among which are the best paper award in the IEEE ASAP conference in 2008 and the Turkish Scientific and Technological Council Encouragement Award for Research in 2009. His research interests include analog and mixed signal integrated circuit design and design automation, especially for analog circuits.

Michael Eick (*michael.eick@mytum.de*) received the Dipl.-Ing. Degree (equivalent to M.Sc.) in Electrical Engineering and Information Technology from Technische Universität München, Munich, Germany in 2008. Since 2008, he is working as a research and teaching assistant at the Institute for Electronic Design Automation, Technische Universität München, Munich, Germany. He is currently working toward his Ph.D. His research interests cover the automatic analysis of circuit structures of analog circuits with application to layout and sizing.

Reem El-Adawi (*Reem_ElAdawi@mentor.com*) received the B.Sc. and M.Sc. degrees in 1993 and 2001, respectively, from Ain Shams University, Egypt. Since 1995, she is with Mentor Graphics Inc., Egypt, where she has held various positions. She is currently Engineering Manager for the team responsible for the development of Chameleon ART, an analog retargeting tool.

Francisco V. Fernández (*pacov@imse-cnm.csic.es*) got the Physics-Electronics degree from the University of Seville in 1988 and his Ph.D. degree in 1992. In 1993, he worked as a postdoctoral research fellow at Katholieke Universiteit Leuven (Belgium). From 1995 to 2009, he was an Associate Professor at the Dept. of Electronics and Electromagnetism of University of Seville, where he was promoted to full professor in 2009. He is also a researcher at CSIC-IMSE-CNM. His research interests lie in the design and design methodologies of analog and mixed-signal circuits. Dr. Fernández has authored or edited three books and has co-authored more than 100 papers in international journals and conferences. Dr. Fernández is currently the Editor-in-Chief of *Integration, the VLSI Journal* (Elsevier). He regularly serves at the Program Committee of several international conferences. He has also participated as researcher or main researcher in several National and European R&D projects.

Jan Freuer (*jan.freuer@de.bosch.com*) received his diploma degree in Computer Science from the Eberhard-Karls University in Tübingen, Germany in 2002. Since

2004, he has been with the R&D department AE/EIM of the automotive electronics division of Robert Bosch GmbH in Reutlingen, Germany. He is currently finalizing his Ph. D. degree in computer science at the Carl von Ossietzky University in Oldenburg, Germany. His main research interests are related to constraint-engineering topics for analog design automation.

Helmut Graeb (*graeb@tum.de*) got his Dipl.-Ing., Dr.-Ing., and habilitation degrees from Technische Universität München in 1986, 1993, and 2008, respectively.

He was with Siemens Corporation, Munich, from 1986 to 1987, where he was involved in the design of DRAMs. Since 1987, he has been with the Institute of Electronic Design Automation, TUM, where he has been the head of a research group since 1993.

He has published more than 100 papers, six of which were nominated for best papers at DAC, ICCAD, and DATE conferences. His research interests are in design automation for analog and mixed-signal circuits, with particular emphasis on Pareto optimization of analog circuits considering parameter tolerances, analog design for yield and reliability, hierarchical sizing of analog circuits, analog/mixed signal test design, semidiscrete optimization of analog circuits, structural analysis of analog and digital circuits, and analog placement.

Dr. Graeb has, for instance, served as a Member or Chair of the Analog Program Subcommittees of the ICCAD, DAC, and D.A.T.E conferences, as Associate Editor of the *IEEE Transactions on Circuits and Systems Part II: Analog and Digital Signal Processing* and *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, and as a Member of the Technical Advisory Board of MunEDA GmbH Munich. He is Senior Member of IEEE (CAS) and member of VDE (ITG).

He was the recipient of the 2008 prize of the Information Technology Society (ITG) of the Association for Electrical, Electronic, and Information Technologies (VDE), of the 2004 Best Teaching Award of the TUM EE Faculty Students Association, of the 3rd prize of the 1996 Munich Business Plan Contest.

Göran Jerke (*goeran.jerke@ieee.org*) received his diploma degree in Electrical Engineering from the Dresden University of Technology in Dresden, Germany. Since 2000, he is with the R&D department AE/EIM of the automotive electronics division of Robert Bosch GmbH in Reutlingen, Germany. His research work focuses on analog design automation and reliability-aware physical design of ICs where he has published numerous papers. In 2004, he received the EDA Achievement Award from the German edacentrum e. V. He is currently finishing his Ph.D. degree in Electrical Engineering at the Dresden University of Technology.

Jens Lienig (*jens.lienig@ifte.de*) received the M.Sc. (diploma), Ph.D. (Dr.-Ing.) and habilitation degrees in Electrical Engineering from Dresden University of Technology, Dresden, Germany, in 1988, 1991, and 1996, respectively. He is currently a Full Professor of Electrical Engineering at Dresden University of Technology, where he is also Director of the Institute of Electromechanical and Electronic Design. From 1999 to 2002, he worked as Tool Manager at Robert Bosch GmbH in Reutlingen, Germany, and from 1996 to 1999, he was with Tanner Research Inc. in

Pasadena, CA. From 1994 to 1996, he was a Visiting Assistant Professor with the Department of Computer Science, University of Virginia, Charlottesville, VA. From 1991 to 1994, he was a Postdoctoral Fellow at Concordia University in Montréal, PQ, Canada. His current research interests are in the physical design automation of VLSI circuits, MCMs, and PCBs, with a special emphasis on electromigration, 3D design, and constraint-driven design methodologies. Prof. Lienig has served on the Technical Program Committees of the DATE, SLIP, and ISPD conferences. He is a Senior Member of IEEE.

Mark Po-Hung Lin (*marklin@ccu.edu.tw*) received the B.S. and M.S. degrees in Electronics Engineering from National Chiao Tung University (NCTU), Hsinchu, Taiwan, in 1998 and 2000, respectively, and the Ph.D. degree in the Graduate Institute of Electronics Engineering (GIEE), National Taiwan University (NTU), Taipei, Taiwan, in 2009.

From 2000 to 2007, he was with Springsoft, Inc., Hsinchu, Taiwan, where he was involved in the design of automatic schematic generation in both *Verdi*^{TM} Automated Debug System and *Laker*^{TM} Advanced Design Platform, and the design of custom placement and routing in *Laker*^{TM} Custom Layout System. In 2008, he was a visiting scholar in the Department of Electrical and Computer Engineering, University of Illinois at Urbana-Champaign (UIUC), IL, USA. He is currently an Assistant Professor of the Department of Electrical Engineering, National Chung Cheng University, Chiayi, Taiwan. His current research interest lies in electronic design automation, with an emphasis on physical design for analog IC's.

Elisenda Roca (*eli@imse-cnm.csic.es*) received the physics and the Ph.D. degrees from the University of Barcelona, Spain, in 1990 and 1995, respectively. From November 1990 to April 1995, she worked at IMEC, Leuven, Belgium, in the field of infrared detection aiming to obtain large arrays of CMOS compatible silicide Schottky diodes. Since 1995, she has been with the Institute of Microelectronics of Seville, (IMSE-CNM-CSIC), Spain, where she holds the position of Tenured Scientist. Her research interests are centered in the design of CMOS Vision Systems on-Chip (VSoC) with integrated high dynamic range sensors, and modeling and design methodologies for analog integrated circuits. She has been involved in several research projects with different institutions: Commission of the EU, ESA, ONR-NICOP, etc. She has also co-authored more than 50 papers in international journals, books, and conference proceedings.

Hazem Said (*Hazem.Said@eng.asu.edu.eg*) received the B.Sc. and M.Sc. degrees in Computer Engineering from Ain Shams University, Egypt, in 1999 and 2006, respectively. He is currently pursuing his Ph.D. degree in electrical engineering at the same University in the field of fast electrical simulation. He worked as a Senior Development Engineer at Mentor Graphics Inc., Egypt, on automatic analog design migration from 2002 to 2007. His research interests include Computer Algorithms, Electronic CAD Algorithms, Artificial Intelligence, and Quantum Computing.

Ulf Schlichtmann (*ulf.schlichtmann@tum.de*) received the Dipl.-Ing. and Dr.-Ing. degrees in Electrical Engineering and Information Technology from Technische

Universität München (TUM), Munich, Germany, in 1990 and 1995, respectively. From 1994 to 2003, he was with the Semiconductor Group of Siemens AG which, in 1999, became Infineon Technologies AG. There he held various technical and management positions in design automation, design libraries, IP reuse, and product development. Since 2003, he has been with TUM as Professor and Head of the Institute for Electronic Design Automation. His research interests are in computer-aided design of electronic circuits and systems, with special emphasis on designing robust systems. Since 2008, he serves as Dean of TUM's department of Electrical Engineering and Information Technology.

Hussein Shahein (*hussein.shahin@eng.asu.edu.eg*) obtained his Ph.D. from the Moore School of Electrical Engineering, University of Pennsylvania, PA, USA, in 1972. He was a postdoctoral Fellow with the University of Pennsylvania Computer Center (Uni-Coll). He also worked at the IBM research Laboratories in San Jose, CA, USA. He then joined the Computer and Systems Engineering Department, Ain Shams University, Egypt. Prof. Shahein served as visiting Professor at King AbdulAziz University, Jeddah, KSA, the Computer Science Dept., University of Bahrain, and the Electrical Engineering Department, UAE University, Al Ain, where he was the Department Chair. He also served as the Chair of the Computer and Systems Engineering Department, Ain Shams University, Egypt, and the Dean of the Faculty of Computer Science, Misr International University, Egypt. He is the Chairman of the Board of Elmohandis Information Systems Company. Prof. Shahein has supervised many Ph.D. and M.Sc. students. His research interests include Optimization, Computer Networks, Networks and data Security.

Martin Strasser (*strasser@tum.de*) received his Dipl.-Ing. degree in Electrical Engineering and Information Technology from Technische Universität München (TUM), Munich, Germany, in 2005. From 2003 to 2005, he was with MunEDA GmbH, before he joined the Institute for Electronic Design Automation at TUM. In this institute, he is currently working toward his Dr.-Ing. degree. His research interests are in electronic design automation for analog circuits, especially the automation of the layout generation.

Ahmet Unutulmaz (*unutulm@boun.edu.tr*) received his B.S. and M.S. degrees in Electrical Engineering from Boğaziçi University, Istanbul, Turkey in 2008. Since 2008, he has been working as a research and teaching assistant at the Department of Electrical Engineering, Boğaziçi University, Istanbul, Turkey. His research interests include automatic synthesis of analog layouts and integration of design automation tools.

Contents

Part I Placement

1 Device-Level Topological Placement with Symmetry Constraints 3
Florin Balasa

2 Hierarchical Placement with Layout Constraints 61
Mark Po-Hung Lin and Yao-Wen Chang

3 Deterministic Analog Placement by Enhanced Shape Functions 95
Martin Strasser, Michael Eick, Helmut Graeb, and Ulf Schlichtmann

Part II Routing

4 Routing Analog Circuits 149
Günhan Dündar and Ahmet Unutulmaz

Part III Layout in the Design Flow

5 Analog Layout Retargeting 205
Hazem Said, Mohamed Dessouky, Reem El-Adawi, Hazem Abbas, and Hussein Shahein

6 Closing the Gap Between Electrical and Physical Design: The Layout-Aware Solution 243
Rafael Castro-López, Elisenda Roca, and Francisco V. Fernández

7 Constraint-Driven Design Methodology: A Path to Analog Design Automation 269
Göran Jerke, Jens Lienig, and Jan B. Freuer

Index 299

Part I
Placement

Chapter 1
Device-Level Topological Placement with Symmetry Constraints

Florin Balasa

Abstract The traditional way of approaching placement problems in computer-aided design (CAD) tools for analog layout is to explore an extremely large search space of feasible or unfeasible placement configurations (called flat representations of the layout), where the cells are moved in the chip plane by a stochastic optimizer – like simulated annealing or a genetic algorithm.

This chapter discusses the possible use in analog placement problems with symmetry constraints of topological representations of the layout, encoding systems that are not restricted to slicing floorplan topologies. First, the chapter gives an overview of several data structures that may be used in the evaluation of various topological representations of the layout – therefore, in building the placement from the layout encoding. Afterwards, the chapter presents a subset of sequence-pairs – called "symmetric-feasible" – that allows to take into account the presence of an arbitrary number of symmetry groups of devices during the exploration of the solution space. Alternatively, the possible use of tree representations instead of "symmetric-feasible" sequence-pairs is also discussed.

The computation times exhibited by the topological approaches are significantly better than those of the placement algorithms using the traditional exploration strategy based on flat representations, while preserving a similar quality of the placement solutions.

1.1 Introduction

1.1.1 CAD for Analog Layout

In recent years, complete systems that used to occupy one or more boards have been integrated on a few chips or even on a single chip. Examples of such *systems-on-a-chip* (SoC's) are networking interfaces, wireless designs, or new

F. Balasa (✉)
Department of Computer Science and Information Systems, Southern Utah University, Cedar City, UT 84721, USA
e-mail: balasa@suu.edu

H.E. Graeb (ed.), *Analog Layout Synthesis: A Survey of Topological Approaches*,
DOI 10.1007/978-1-4419-6932-3_1, © Springer Science+Business Media, LLC 2011

generations of integrated telecommunication systems – that include analog, digital, and eventually, radiofrequency (RF) sections on one chip. Although most functions in such integrated systems are implemented with digital or digital signal processing circuitry, the analog circuits needed at the interface between the electronic system and the real world are now being integrated on the same die for reasons of cost and performance.

In the digital domain, computer-aided design (CAD) tools are fairly well developed, especially for the lower level of the design flow. Unlike analog circuits, a digital system can naturally be modeled in terms of Boolean representations and programming language constraints; its functionality can easier be represented in algorithmic form. Consequently, many lower-level aspects of the digital design process are fully automated. Research interests are now moving in the direction of system synthesis, where system-level specifications are translated into hardware–software co-architecture. The level of automation is far from the "push-button" stage, but the advance of CAD tools is keeping up reasonably well with the progress of technology.

Unfortunately, the situation is worse on the analog side. Apart from circuit simulators, layout editing environments, or layout verification tools, real commercial solutions are only beginning to appear as the result of a valuable research and development (R&D) effort in the field [1–3]. Some of the main reasons for this lack of automation are that analog design in general is less systematic and more heuristic in nature than digital design, requiring specialized knowledge, design skills, and years of experience; analog circuits are more sensitive to parasitic disturbances, crosstalk, substrate noise, supply noise, etc.; in addition, the variety of schematics and diversity of device sizes and shapes are much larger. These differences from digital design explain why specific analog solutions need to be developed. Due to the lack of mature, robust analog CAD tools, analog designs today are still largely being handcrafted, with limited CAD support available (except simulators, interactive layout environments). The design cycle for analog (and mixed-signal) IC's remains long and error prone.

The physical implementation step in the analog design flow corresponds to a variety of tasks that can be grouped into two major areas: (a) analog circuit-level (or block-level) layout synthesis, which has to transform a sized transistor-level schematic into a mask layout, and (b) system-level layout assembly, in which the basic functional blocks are already laid out and the goal is to floorplan, place, and route them, as well as to distribute the power and ground connections. These two areas are also interleaved as most design flows require a mix of top-down and bottom-up approaches. This chapter will address placement issues in the field of block-level layout synthesis.

The optimization-based place-and-route layout generation approaches consist of synthesizing the layout solution by optimization techniques according to some cost functions. They differ from the earlier procedural module generation techniques [4], in which the layout of the entire circuit is precoded in a software tool that generates the complete layout for the actual parameter values entered at run time. Also, they differ from the related set of template-driven methods [5], where a geometric

template fixing the relative position and interconnection of the devices is stored for each circuit. The advantages of the optimization-based approaches are their generality and flexibility in terms of performance and area. The penalty to pay is they require a more significant computational effort; also, the layout quality is more dependent on the algorithms, on the cost functions employed, on providing a complete set of design constraints and taking them into account during the optimization.

1.1.2 The Device-Level Analog Placement Problem

The decision whether a given set of fixed-oriented rectangles, having widths and heights real numbers, could be packed onto a chip of known width and height was proven to be NP-complete [6], while the problem of finding a minimum area packing was shown to be NP-hard. Like many other VLSI placement problems – for instance, chip floorplanning and macro cell digital placement – the analog placement must also cope with optimally packing arbitrarily sized modules.

In addition to that, a placement tool must include specific capabilities to automatically produce analog device-level layouts matching in density and performance the high-quality manual layouts. Such specific features are, for instance, (1) the ability to deal with topological constraints for symmetry and device matching; (2) the ability to arrange devices such that critical structures are shared – design technique known as *device merging* or *geometry sharing* [7], aiming to reduce both layout density and induced parasitics; (3) the existence of a (built-in) library of predefined module generators and the ability to exploit their reshaping capabilities during the placement process [1].

1.1.3 Overview of Analog Placement Methods

Due to the complexity of the basic problem, several heuristic placement techniques have been attempted first. The *constructive* approaches consist in evolving gradually the placement solution by selecting one module at a time and positioning it in the "best" available location. Several systems for analog placement employed constructive methods: Kayal et al. developed an expert knowledge base to guide the placement [8]; Mehranfar suggested a schematic-driven approach, using a constructive scheme based on connectivity and relative positioning in the input schematic [9, 10]. The constructive methods are fast, scaling well with the problem size; their basic drawback is the dependence on the selection order of devices. Lacking a global view in dealing with a variety of interacting quality measures, this strategy yields sometimes poor placement solutions. A technique achieving a better global optimization of the device positions – by iteratively combining min-cut partitioning and force-directed placement – has been employed in an interactive environment for full-custom designs [11].

Other class of methods translates an analog placement problem into a constrained (combinatorial) optimization. Earlier techniques extracted mainly (hard and soft) nonquantitative constraints for the subsequent optimization phase [12]. In later approaches, the optimization was *performance-driven*, doing a quantitative evaluation (based on estimation models) of the placement solutions, to ensure the performance of the final layout [13, 14].

As combinatorial optimization engines, the simulated annealing [15] and genetic algorithms [16] were effective choices for solving industrial analog placement problems. These algorithms use stochastically controlled "hill-climbing" to avoid being trapped in local minima during the optimization process. In addition, they do not impose severe constraints on the size of the problems or on the mathematical properties of the cost function – like most optimization algorithms in mathematical programming. While efficiently trading-off between a variety of layout factors – such as area, total net length, aspect ratio, maximum chip width and/or height, cell orientation, "soft" cell shape, etc. – they support incremental addition of new functionality (for instance, updates of cost function and/or constraints) and they are relatively easy to implement (although good tuning needs more time). This is why simulated annealing, the most mature of the stochastic techniques, provided the engine for effective software packages both in digital (TimberWolfSC v7.0 [17]) and in analog design: ILAC [18], KOAN/ANAGRAM II [7, 12] – that evolved into the *NeoLinear* system, PUPPY-A [13], LAYLA [14]. More recently, a two-phase approach using both a genetic algorithm and simulated annealing with dynamic adjustment of the parameters has been reported [19, 20]. Another recent technique derives linear inequalities from constraint graphs extracted from sequence-pairs, and obtains the placement by linear programming within a simulated annealing framework [21].

While this chapter will focus on optimization techniques for analog placement, other effective analog layout tools are *template-driven* [22, 23]. These tools are built on template databases containing analog circuits designed by experienced experts. Upon the arrival of a new design demand, the system selects a suitable template from the database, adding information on the target technology, design rules, device sizes, etc., to re-generate automatically the target layout.

1.1.4 Placement for Layout Symmetry

In high-performance analog circuits, it is often required that groups of devices are placed symmetrically with respect to one or several axes. Differential circuit techniques are used extensively to improve the accuracy, power supply rejection ratio, and dynamic range of many analog circuits. The full performance potential of many of these circuits cannot be achieved unless special care is taken to match the layout parasitics in the two halves of the differential signal path. Failure to match these parasitics in, for instance, differential analog circuits can lead to higher offset voltages

and degraded power-supply rejection ratio [7]. The main reason of symmetric placement (and routing, as well) is to match the layout-induced parasitics in the two halves of a group of devices.

Placement symmetry can also be used to reduce the circuit sensitivity to thermal gradients. Some VLSI devices (the bipolar devices, in particular) exhibit a strong sensitivity to ambient temperature. If two such devices are placed randomly relative to the isothermal lines, a temperature-difference mismatch may result. Failure to adequately balance thermal couplings in a differential circuit can even introduce unwanted oscillations [24]. To combat potentially induced mismatches, the thermally sensitive device couples should be placed symmetrically relative to the thermally radiating devices. Since the symmetrically placed sensitive components are equidistant from the radiating component(s), they see roughly identical ambient temperatures and no temperature-induced mismatch results.

It is more often the case that a circuit has a mix of symmetric and asymmetric components. For example, the two-stage Miller compensated opamp shown in Fig. 1.1 has a symmetric differential input stage, but it has an asymmetric single-ended output stage.

The typical forms of symmetry which should be handled by an analog placement tool are [7]:

1. Mirror symmetry: Consists in placing a symmetry group of cells about a common axis such that the cells in every pair have identical geometry and mirror-symmetric orientation. It is the most standard form of layout symmetry. There are two major advantages of this placement arrangement. First, because sibling devices are forced to adopt identical geometry, device-related

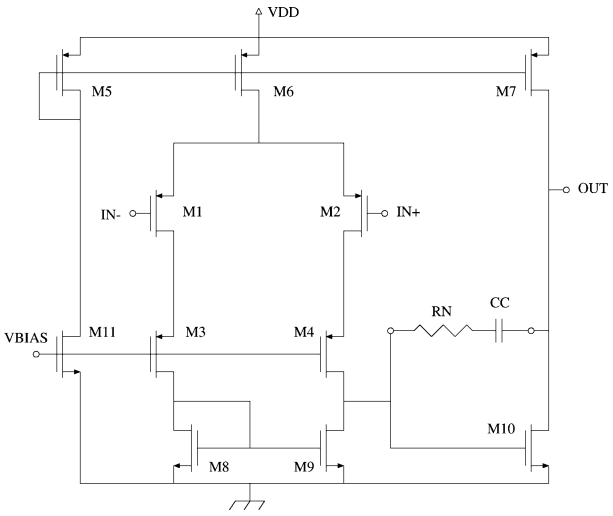


Fig. 1.1 Schematic of a two-stage Miller compensated opamp with asymmetric output stage

parasitics are balanced and device matching characteristics are improved. Second, mirror-symmetric placement aligns device terminals in a way that makes mirror-symmetric routing[1] possible.

2. Perfect symmetry: Differs from the previous by the identical (rather than mirror-symmetric) orientations of the paired devices. This type of symmetry is sometimes required in order to meet very stringent matching requirements. When there is a possibility of anisotropic fabrication disturbances (e.g., oblique-angle ion implantation) [7], the best matching is achieved when paired devices are placed in identical orientations. Perfectly symmetric placement presents a difficult layout problem: because the device terminals are no longer mirror-symmetric, one cannot use mirror-symmetric routing to connect sibling devices with parasitic matched wires. Instead, one has to route parasitic matched wires, which are not geometrically symmetric. This can be particularly difficult when there is a mix of symmetric and asymmetric circuitry.
3. Self-symmetry: Characteristic for devices presenting a geometrical symmetry and sharing the same axis with other pairs of symmetric devices. Self-symmetric devices have two uses. First, it is often desirable to place asymmetric devices (e.g., devices in bias networks) in the middle of a mirror-symmetric layout. This greatly simplifies wiring in the case that the device is highly connected to devices on both sides of the symmetric signal path. Such an arrangement presents mirror-symmetric terminals to the left and right halves of the circuit, so that they can participate in mirror-symmetric routing. Second, self-symmetry is useful in creating thermally symmetric layouts.

A subset of cells is called a *symmetry group* if all cells are exhibiting a form of symmetry and, in addition, they all share a common symmetry axis. The symmetry constraints for a pair of devices (B_i, B_j) in the kth symmetry group have the form: $(x_i + w_i) + x_j = 2 \cdot x_{\text{symAxis}_k}$ and $y_i = y_j$, where (x_i, y_i) are the left-bottom coordinates of device B_i, w_i denotes its width, and x_{symAxis_k} is the abscissa of the symmetry axis of the kth group (assuming the axis is vertical). Similarly, a self-symmetric device B_i must satisfy the constraint: $x_i + w_i/2 = x_{\text{symAxis}_k}$. In this chapter, the symmetry axes will be considered vertical since this is the typical way most layouts are designed.

1.1.5 The Absolute Representation of the Layout

A combinatorial optimization algorithm for solving placement problems can equally operate with two distinct spatial representations of the placement configurations. The earliest is the so-called *absolute* (or *flat*) representation introduced by Jepsen and Gellat [25] in a macro-cell placement tool. In this representation, the cells

[1] Mirror-symmetric routing assumes that paired nets be implemented using geometrically mirror identical wire segments.

are specified in terms of *absolute* coordinates on a plane. The moves are simple translations (coordinate shifts) or changes in cell orientation – rotations and mirror operations. The cells are allowed to overlap even in illegal ways,[2] as no restriction is made referring to the relative position of a cell with respect to another cell. A penalty cost term – typically, quadratic – is associated with the total illegal overlap, and this penalty must be driven to zero during the minimization of the cost function. The flat representation is well-suited to handle device matching and symmetry constraints – typical to analog layout – since they are easy to model and maintain during successive moves; it also allows to explore the beneficial device overlaps. For these reasons and also its inherent simplicity, the absolute representation was the choice for KOAN/ANAGRAM II [12], PUPPY-A [13], and LAYLA [14] systems.

However, this representation has also shortcomings explained, for instance, in [17]. First, the optimization process is slow: since the exploration space is very large, many moves can yield a small decrease of the cost function. Second, the total illegal overlap (representing only one term of the cost function) is not necessarily equal to zero in the final placement solution: a post-processing step aiming to eliminate the gaps and overlaps must be performed, affecting even more the computation time and degrading the solution optimality. Moreover, the *weight* of the overlap term in the cost function must be carefully chosen: if it is too large, the search ability of the optimizer for a good placement (in terms of area, total net length, etc.) may be impeded; if it is too small, the cells may have the tendency to collapse since the importance of illegal overlaps is small. To combat this effect, an earlier version of the TimberWolf system [17] used a sophisticated negative control scheme to determine the optimum values of the cost term weights.

The flat representation approach trades off a larger number of moves for easier and quicker to build layout configurations – which may not be always physically realizable though. On the other hand, a second class of placement representations – named *topological* – allows to trade off more complex (but physically correct!) layout constructions per each move of the optimization engine against a smaller number of moves.

1.1.6 Topological Representations of the Layout

Different from the flat representation where the cell positions are specified in terms of their coordinates, in a topological representation a placement configuration is *encoded*: the cell positions are *relatively* specified, based on topological relations between cells. The first popular representations were employing the so-called *slicing model*, introduced by Otten [26]. In this model, the cells are organized in a set of slices, which recursively bisect the layout horizontally and vertically. The direction

[2] In analog layout, cells can overlap not only in *legal* but also *beneficial* ways ("device merging" or "geometry sharing" [12]).

and nesting of the slices is recorded in a *slicing tree* or, equivalently, in a *normalized Polish expression* [27]. The annealing algorithm (as a typical optimization engine) does not move explicitly the cells – as in the flat representation: the moves are *modifications of the placement codes* (for instance, small reorganizations of a slicing tree, or small changes in a Polish expression that preserve the properties of the encoding). These moves alter *indirectly* the relative positions of the cells. In topological representations, cells cannot overlap illegally, which may lead to an improved efficiency in the placement optimization.

However, the slicing model limits the set of reachable layout topologies. This can degrade layout density, especially when cells are very different in size, which is often the case in analog layout. Furthermore, symmetry and matching constraints are difficult to maintain between successive moves: for instance, a slicing-style placement tool had to implement symmetry constraints in the cost function through the use of *virtual* symmetry axes [28] – a less efficient solution. Although the ILAC system [18] employed slicing trees, it is widely acknowledged today that this model is not a good choice for high-performance analog layouts.

After 1995, several novel topological representations, not restricted to slicing floorplan topologies, have been proposed. A remarkably elegant encoding system was proposed by Murata et al., who suggested to encode the "left-right" and "above-below" topological relations using two sequences of cell permutations (see Sect. 1.3 for more details), named a *sequence-pair* [29]. A $O(n^2)$ algorithm (n being the number of cells) based on building a pair of horizontal and vertical constraint graphs was used to construct a compact placement from its encoding, operation called *sequence-pair evaluation*. More recently, a different approach – based on the computation of the longest common subsequence in a pair of weighted sequences – was proposed by Tang et al. [30, 31]. The latest evaluation algorithm achieves a $O(n \log\log n)$ complexity [31] using an efficient model of priority queue [32]. Nakatake et al. devised a meta-grid structure without physical dimensions (called *bounded-sliceline grid* or *BSG*) to define the topological relations between blocks. The construction of the placement configuration from a BSG is of quadratic complexity [33].

Guo et al. proposed the *ordered tree* (*O-tree*) data structure to reduce the negative effect of code redundancies from the two previous representations [34]. Independently, Chang et al. [35] and Balasa [36] suggested similar representations based on binary trees. These encodings are based on the *natural correspondence* between forests of rooted trees and binary trees [37]. Due to the one-to-one transformation mentioned above, all these tree representations can be regarded as equivalent.

The *corner block list* (CBL) [38] is a representation that is used to encode *mosaic* floorplans (that is, floorplans with zero dead-space). The transitive closure graph (TCG), introduced by Lin and Chang [39], is based on two directed graphs having a node for each cell; their edges correspond to the horizontal and, respectively, vertical topological relations between cells. Different from the tree representations [34, 35], the sequence-pair, the bounded-sliceline grid, the corner block list, and the transitive closure graphs define the topological relations between cells independent of their dimensions.

1.1.7 Selecting a Topological Representation for Analog Placement

The nonslicing topological representations were initially used in block placement and floorplanning tools [40–42]. Could these representations be successfully used in placement tools for analog layout? Which of the topological representations would be better-suited? At a first glance, the main selection criteria should be the same as in block placement: (1) a representation with a low (or even zero) code redundancy to have an exploration space as reduced in size as possible, and (2) the existence of an efficient code evaluation algorithm (preferably of linear complexity) building as fast as possible the placement configuration from the current code in each inner-loop iteration of the simulated annealing.

Without denying the importance of the above criteria, other features specific to analog layout must be taken into account as well. As already explained in Sect. 1.1.4, many analog designs contain an arbitrary number of symmetry groups of devices (that is, groups of devices having distinct symmetry axes), each group containing an arbitrary number of pairs of symmetric devices with the same geometry, as well as self-symmetric devices – presenting a geometrical symmetry and sharing the same axis with its group. Due to this characteristic, *most of the codes of any topological representation would be infeasible in symmetry point of view*.

In preliminary experiments using sequence-pairs for solving analog placement problems with symmetry constraints [43], a simple exploration scheme was initially attempted: while searching the set of sequence-pairs in a simulated annealing framework, the codes that proved to be infeasible in symmetry point of view during the placement construction were disregarded. Unfortunately, this simple exploration scheme proved to be extremely ineffective, the quality of the placement solutions being very poor. The main reason was revealed to be the huge number of infeasible codes, which were overwhelming in comparison to the "symmetric-feasible" ones.

These preliminary tests showed that symmetry is difficult to model within a topological representation: *how to recognize the codes complying with the given set of symmetry constraints without building the corresponding layout?* Moreover, assuming the current code is symmetric-feasible, how to prevent the annealer to move from it to an infeasible code? Maintaining the "symmetric-feasibility" of the codes during the annealer's moves is, in general, a nontrivial task, specific to the topological representation employed: *how to restrict the exploration only to the subspace of symmetric-feasible codes?*

A topological representation would prove to be a good candidate for solving analog placement problems with symmetry constraints if it possessed a property characterizing codes able to generate placements such that symmetry constraints be satisfied. In the absence of such a property, the fact that a certain representation has an evaluation algorithm of linear complexity is of a lesser importance since most of the codes would be symmetric-infeasible anyway. Such a property would allow to efficiently restrict the exploration to a subspace of "symmetric-feasible" codes.

Several topological exploration techniques for analog placement investigated how to handle symmetry constraints more efficiently. Approaches using sequence-pairs [21, 43], trees[3] [45], and transitive closure graphs [46] have been developed.[4] Notice that the complexity of the code evaluation can be affected when symmetry constraints have to be taken into account. Dealing with an arbitrary number of symmetry groups of devices during the code evaluation necessitates nontrivial algorithmic modifications, paying also a significant computational toll. For instance, the complexity of the evaluation algorithm in [44] is quadratic, while the evaluation algorithm for O-trees in the absence of symmetry constraints is linear [34].

This chapter will present some data structures used in the evaluation of topological representation, followed by a few topological techniques for device-level placement with symmetry constraints.

1.2 Data Structures for Rectilinear Border Contours

This section will give an overview of several data structures that may be used in the evaluation of a given topological representation of the layout. The algorithms in this section are independent of the choice of the topological representation. To emphasize this independence, we shall take into account the horizontal/vertical topological constraints between the cells rather than a certain abstract representation (since the topological constraints are derived from the layout encoding in specific ways that characterize the abstract representation).

1.2.1 Segment Trees

The *segment tree*, originally introduced by Bentley [48], is a data structure mainly employed in computational geometry, designed to handle operations with intervals whose extremes belong to a given set of coordinates. The coordinates of the intervals can be *normalized* by replacing each of them by its rank in their minimum-to-maximum order. Therefore, without any loss of generality, we may consider these coordinates as integers in the range $[0, n]$.

The (complete) segment tree is, basically, a rooted binary tree, where each node v has attached an interval $v.I = [c, d]$ with integer bounds. If $d - c > 1$, then node v has a left and a right descendant – denoted below as $v.left$ and $v.right$ – having associated the intervals $\left[c, \lfloor \frac{c+d}{2} \rfloor\right]$ and, respectively, $\left[\lfloor \frac{c+d}{2} \rfloor, d\right]$. The intervals attached to the nodes are called *standard*, while those pertaining to the leaves and having the length equal to 1 are named *elementary*.

[3] Actually, the algorithm in [44] exploits properties of the ordered tree codes which are *infeasible* in symmetry point of view, to efficiently detect and hence discard them.

[4] Very recently, a CAD system for analog layout, including a topological placement tool that uses the *corner block list* representation [38], has been proposed [47].

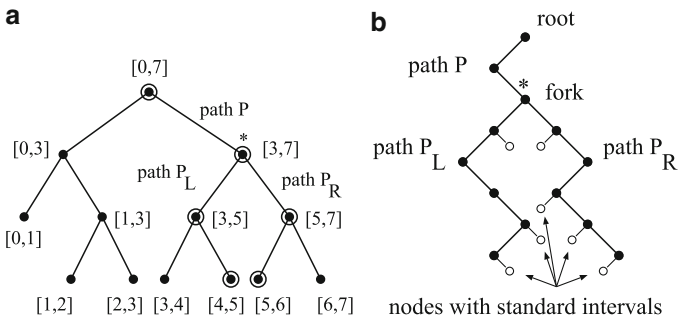


Fig. 1.2 (**a**) Complete segment tree for the root interval [0, 7], and the nodes visited during the insertion of the interval [4, 6]; (**b**) typical tour in the segment tree during the insertion of an interval

A complete segment tree is shown in Fig. 1.2a. This tree is balanced, all the leaves belonging to at most two contiguous levels. The depth of the complete segment tree is $\lceil \log_2(r-l) \rceil$, where $[l, r]$ is the interval attached to the root [49].

The segment tree $T(l, r)$ is designed to store intervals whose extremes belong to the set $\{l, l+1, \ldots, r\}$ in a *dynamic* fashion, that is supporting interval insertions and deletions. The segmentation of an interval $[a, b]$ is completely specified by the operation that stores (inserts) $[a, b]$ in the segment tree $T(l, r)$. To insert an interval, one must visit the nodes in the segment tree along a tour having the following general structure (see Fig. 1.2b): an (possibly empty) initial path P from the root to a node called the *fork* – marked with a star in the figure, from which two (possibly empty) paths P_L and P_R issue. Either the interval being inserted is assigned entirely to the fork (in which case P_L and P_R are both empty), or all the right sons of nodes of P_L, as well as all the left sons of nodes of P_R identify the fragmentation of $[a, b]$ into standard intervals. For instance, Fig. 1.2a shows the nodes visited during the insertion of the interval [4, 6] in the segment tree $T(0, 7)$.

The assignment of an interval to a node v of the segment tree could take different forms, depending upon the requirements of the application. Frequently, all we need to know is the cardinality of the set of intervals assigned to any given node v. This can be managed by a single nonnegative integer data member $v.cnt$, initialized to zero, denoting this cardinality. If this is the case, the assignment of the interval $[a, b]$ to the node v simply becomes $v.cnt = v.cnt + 1$. In other applications, there is need to preserve the identity of the intervals assigned to a node v. Then we may append to each node v a secondary data structure, for instance, a singly linked list, whose records are the identifiers of the intervals. Removing an interval from the segment tree works in a symmetric way. Note that only deletions of previously inserted intervals guarantee correctness.

The segment tree is a versatile data structure with numerous applications. It is extremely used especially in the geometric searching algorithms and the geometry of rectangles [49]. For instance, if one wishes to know the number of intervals containing a given point x, a simple binary search in the segment tree (that is, the traversal of a path from the root to a leaf) readily solves the problem.

In this section, we are going to use the segment tree data structure to compute the device abscissae x_i assuming the device ordinates y_i are already known [50], therefore the extremes of the intervals defining the left and right border contours – the elements of the set $S = \bigcup_i \{y_i,\ y_i + h_i\}$ – are currently fixed. Also, it is assumed that a topological sort of the horizontal constraint graph is available: this order of visiting the nodes ensures that the blocks to the *left* are visited before the blocks to the *right* and, therefore, the horizontal topological constraints would be satisfied.

During the visit of the topologically sorted nodes, a segment tree data structure will be gradually built. The creation of the segment tree is done in a top-down manner, starting with the root and expanding the tree till the nodes associated with elementary intervals. In our application, each node v of the segment tree has attached an interval $v.I$ and a value $v.x$ used for the computation of the cell abscissae x_i. After each iteration, the segment tree will represent the contour of the *right* border of the (partial) placement configuration (see the illustrative example towards the end of this section).

First, the y-coordinates of the devices are "normalized": after sorting them increasingly (and eliminating the duplicate values), the y-coordinates are replaced by their indexes (ranks) in the ordered sequence. Note that the algorithm operates with intervals $[a_i, b_i]$ rather than $[y_i, y_i + h_i]$, where $a_i,\ b_i$ are the indices of y_i and, respectively, $y_i + h_i$ in S – the increasingly-sorted set of the interval endpoints. In this way, the size of the segment tree will be kept minimal and, without loss of generality, the y-coordinates can hence be considered integers in the range $[0, n]$ (n being the number of devices). Note also that the indices $a_i,\ b_i$ can be determined while sorting the set S without affecting the complexity of the sorting operation.

Algorithm: Computation of the device abscissae (x_i) using a segment tree

`let` $x_i = 0$; // reset all the abscissae of the left-bottom corners of the devices
`sort increasingly the set` $S = \bigcup_i \{y_i,\ y_i + h_i\}$;
    // the duplicate elements of the set are eliminated during sorting
`let` m `be the number of elements of set S;`
`SegmentTreeNode` v_0 = *CreateNode* $([0, m-1], 0)$;
    // create the root v_0 of the segment tree
for `each cell` B_i (visited in the order of the topological sort)
  `let` a_i `be the index of` y_i, `and`
  `let` b_i `be the index of` $y_i + h_i$ `in set` S;
  UpdateSegmentTree $(v_0, [a_i, b_i])$;
  UpdateRightContour $(v_0, [a_i, b_i])$;
end_for
$W = \max\{v.x\}$, $\forall v \in$ `SegmentTree;`
    // compute the width W of the placement

After the ordinate normalization, the segment tree is recursively built by the procedure *UpdateSegmentTree* (see below). The *CreateNode* procedure constructs and inserts a new node v in the segment tree – the two parameters being the interval $v.I$ and the value $v.x$. The roots of the left and right subtrees of v are denoted *v.left* and, respectively, *v.right*; they are initially NULL. The procedure *UpdateSegmentTree* is

inserting the normalized interval of $[y_i, y_i + h_i]$ – the spanning of block B_i along the y axis – into the segment tree, decomposing it into standard intervals. At the same time, the abscissa x_i of the left-bottom corner of block B_i is computed by taking the maximum over all the values $v.x$ of the nodes with standard intervals.

procedure *UpdateSegmentTree* $(v, [a_i, b_i])$
  if $v.I \subseteq [a_i, b_i]$ **then**
    if $v.x > x_i$ **then** $x_i = v.x$;
  else `let` $v.I = [c, d]$ `and` $mid = \lfloor \frac{c+d}{2} \rfloor$;
    if v `is currently a leaf of the segment tree` **then**
      $v.left$ = *CreateNode* $([c, mid], 0)$;
      $v.right$ = *CreateNode* $([mid, d], 0)$;
    if $a_i < mid$ **then** *UpdateSegmentTree* $(v.left, [a_i, b_i])$;
    if $mid < b_i$ **then** *UpdateSegmentTree* $(v.right, [a_i, b_i])$;
end_procedure

The procedure *UpdateSegmentTree* visits the nodes in the segment tree along a tour having the general structure shown in Fig. 1.2b. Subsequently, the procedure *UpdateRightContour* sets the values of all the nodes corresponding to these standard intervals to $x_i + w_i$ – the abscissa of B_i's right border.

procedure *UpdateRightContour* $(v, [a_i, b_i])$
  if $v.I \subseteq [a_i, b_i]$ **then** $v.x = x_i + w_i$;
  else `let` $v.I = [c, d]$ `and` $mid = \lfloor \frac{c+d}{2} \rfloor$;
    if $a_i < mid$ **then** *UpdateRightContour*$(v.left, [a_i, b_i])$;
    if $mid < b_i$ **then** *UpdateRightContour*$(v.right, [a_i, b_i])$;
end_procedure

The computation of each cell abscissa, based on the decomposition of the normalized interval $[y_i, y_i + h_i]$ into standard intervals, is followed by an update of the values $v.x$ of the visited nodes. To avoid performing any computation twice, the decomposition into standard intervals can be done top-down, using two stacks to store the visited nodes, one for the standard nodes – the "white" nodes in Fig. 1.2b, the other for the nodes on the paths P, P_L, and P_R – the "black" nodes in the same figure. Then the update of the values $v.x$ can be easily done bottom-up. Hence, the implementation of the procedure *UpdateRightContour* can be performed more efficiently than the recursive version given above for reason of clarity.

The decomposition of the root segment into standard intervals is done in $O(\log n)$ time since the height of the segment tree is at most $\lceil \log_2(m-1) \rceil$ (the root interval being $[0, m-1]$), hence upper-bounded by $\lceil \log_2 n \rceil$. This entails the same complexity for the procedures *UpdateSegmentTree* and *UpdateRightContour*. Since the sorting of the set S, together with the computation of the indices a_i and b_i, take $O(n \log n)$ time, the overall complexity of the algorithm computing the device abscissae is thus $O(n \log n)$.

Example. Consider a layout with nine rectangular blocks having the widths and heights indicated: A(140 × 30), B(40 × 20), C(50 × 50), D(20 × 60),

E(20 × 60), F(20 × 30), G(50 × 60), H(20 × 10), and I(40 × 20). Assume the cell ordinates are known or have been previously determined: $[y_A\ y_B \ldots y_I] = [0\ 40\ 60\ 30\ 30\ 30\ 60\ 30\ 40]$. Let us assume that the order of the nodes in the topological sort of the horizontal constraint graph is alphabetical: A,B, ..., I. This example illustrates the computation of the device abscissae using a segment tree.

All the cell abscissae are initially zero. After the sorting and elimination of duplicates, the set $S = \bigcup_{i=1}^{i=9}\{y_i,\ y_i + h_i\} = \{0, 30, 40, 60, 90, 110, 120\}$ has $m = 7$ elements. Due to the normalization, the root node v_0 of the segment tree has associated the interval [0, 6]. (By normalization, the depth of the segment tree is reduced from $\lceil \log_2 120 \rceil = 7$ to $\lceil \log_2 6 \rceil = 3$.) The interval $[y_A, y_A + h_A] = [0, 30]$ of the first node visited in the preorder traversal is normalized to $[a, b] = [0, 1]$ (since the indices of the elements 0 and 30 in S are 0 and, respectively, 1). As $v_0.I = [0, 6]$ and $mid = 3$, two new nodes v_1 and v_2 are created having attached the intervals [0, 3] and [3, 6]. *UpdateSegmentTree*$(v_1, [0, 1])$ is then recursively called and two new nodes v_3 and v_4 – having the intervals [0, 1] and [1, 3] – are created (see Fig. 1.3a). The execution of *UpdateSegmentTree*$(v_0, [0, 1])$ yields $x_A = v_3.x = 0$ since $v_3.I = [0, 1]$. Afterward, *UpdateRightContour* will visit once again the same nodes in the segment tree to update the values of the nodes. In this case, the only node is v_3 (which is actually the "fork," the paths P_L and P_R being empty): therefore, $v_3.x = x_A + w_A = 140$.

Figures 1.3a–i display the segment tree after the insertion of each normalized y-spanning intervals $[a_i, b_i]$ in the segment tree, the cells being successively placed in the order given by the topological sort. The nodes corresponding to the standard segments are represented as double circles, while the "fork" nodes are marked with a star. The computation of the cell abscissae yields successively: $[x_A\ x_B \ldots x_I] = [0\ 0\ 0\ 50\ 70\ 90\ 90\ 110\ 110]$, and the final value of the root $v_0.x = 150$ is the current width of the analog block. The placement corresponding to the last segment tree in Fig. 1.4a is shown in Fig. 1.4b.

Although in this illustrative example the final segment tree is complete, that is, all the leaves have attached elementary intervals, this is not always the case – which is quite desirable for the practical running times. □

When this algorithm is embedded into a combinatorial optimization framework, like simulated annealing, it is not efficient to create a new segment tree for every inner-loop iteration of the annealer. Actually, the segment tree should be created only once, at the beginning of the annealing process. A *re-initialization* of the segment tree at the beginning of each inner-loop iteration would suffice. The rationale of this procedure is explained below.

The segment tree is used in our context for the computation of the abscissae of n devices, the largest interval associated with the root being $[0, n]$. At each iteration of the simulated annealing, the root interval will be of the form $[0, k]$, where $k \leq n$. Since all these intervals are included in $[0, n]$, there is no need to build a new segment tree for every evaluation of the topological representation (although the root interval $[0, k]$ is typically changing after each iteration of the annealer). In fact, it is sufficient to build only once a segment tree having the root interval $[0, n]$. Indeed, since

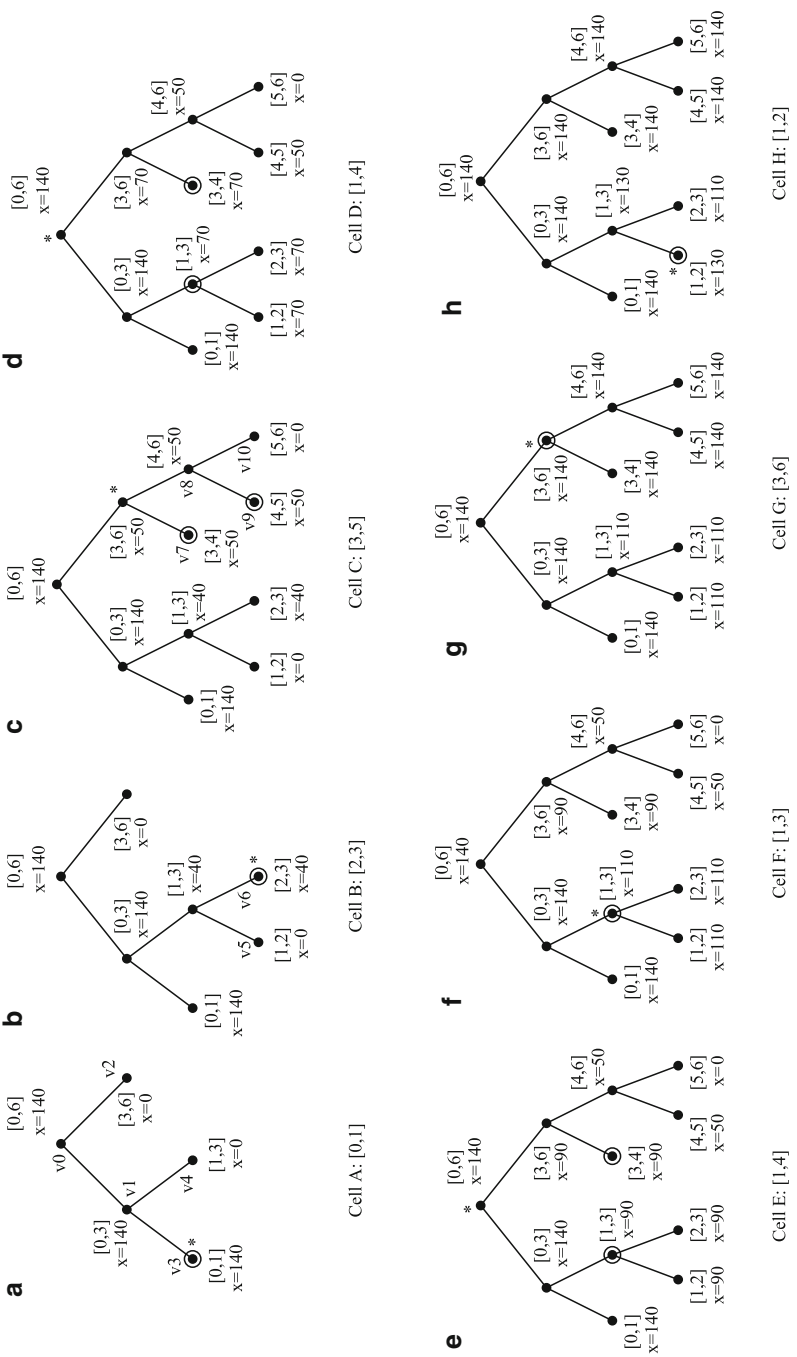


Fig. 1.3 **(a–h)** The segment tree evolution

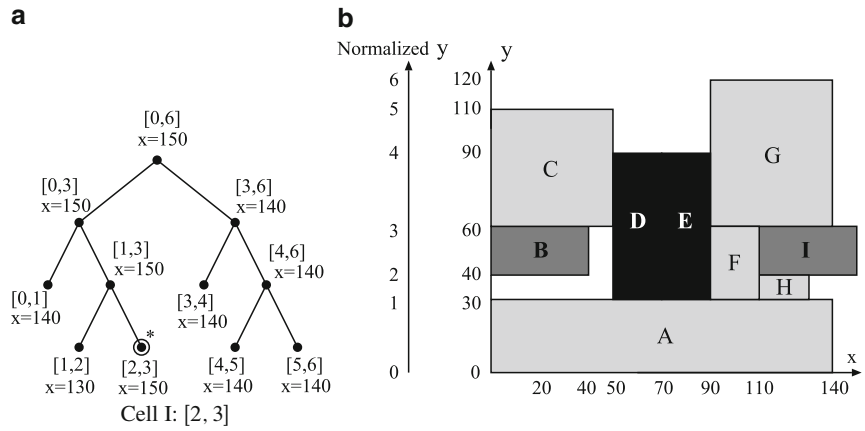


Fig. 1.4 Illustrative example: (**a**) the last segment tree (after the insertion of the normalized y-interval of cell I, that is $[2,\ 3]$), and (**b**) the final device placement

at each iteration the root interval $[0, k] \subseteq [0, n]$, the current segment tree can actually be embedded in the "larger" one having the root interval $[0, n]$, updating only the intervals $v.I$ of the nodes. Flags attached to the nodes (denoted $v.leaf$) are used to indicate the leaf nodes at any moment. This remark reduces significantly the practical computational effort since the creation and deletion of the segment tree nodes actually happen only once – at the beginning and, respectively, at the end of the annealing process, rather than in each inner-loop iteration. Failure to take this remark into account increases the computation time of the evaluation algorithm by 15–20%.

1.2.2 Red–Black Interval Trees

The interval tree is a binary search tree, with each node having associated a closed interval whose interior is disjoint from the intervals of the other nodes, but whose union is a closed interval as well. In our case, the union of the node intervals will always be $[0, H]$, where H is the chip height. In addition, the intervals of the nodes in any left subtree are to the left (on the real line) of the node interval, while the intervals in the right subtree are to the right of the node interval. (Thus, an in order tree traversal of the data structure lists the intervals in sorted order by the low endpoints.)

Moreover, the interval tree is organized as a *red–black tree* [51] – a binary search tree with an extra bit of storage per node: its *color*, which can be either *red* or *black*. (This color convention was introduced by Guibas and Sedgewick [52] who introduced red–black trees.) The reason of this organization is to ensure an amortized time bound [51] of $O(\log n)$ per each update of the data structure. In addition, if a node is red, its children must be black (the *NULL* pointers or references, when there are no children, are also considered black leaves), and every path from a node to a descendant leaf contains the same number of black nodes. An example of a red–black interval tree is displayed in Fig. 1.5. By constraining, the way nodes can

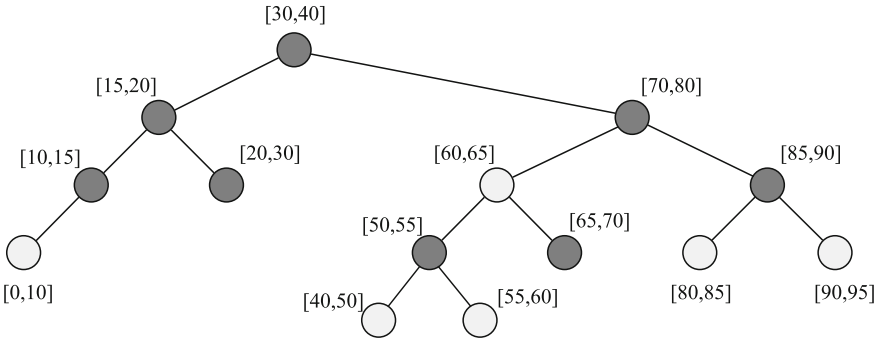


Fig. 1.5 A red–black interval tree (the *red* nodes are shaded, the *black* nodes are darkened)

be colored on any path from the root to a leaf, red–black trees ensure that no such path is more than twice as long as any other [51], so the tree is *approximately* balanced.

The general scheme of the algorithm computing the device abscissae is given below [53]. The assumptions are similar to the ones in Sect. 1.2.1: the device ordinates y_i are already known and a topological sort of the horizontal constraint graph is available. First, the root of the red–black interval tree is created, the node having attached the interval $[0, H]$, where H is the height of the layout. Afterward, the devices are visited in the order of the topological sort, such that the blocks to the *left* are visited before the ones to the *right*, such that the horizontal positioning constraints be satisfied. The red–black interval tree is iteratively updated as a result of the insertion of the new y-spanning interval $[y_i, y_i + h_i]$ of device B_i in the tree.

Algorithm: Computation of the device abscissae (x_i) using a red–black interval tree

`let` $x_i = 0$; // reset all the abscissae of the left-bottom corners of the devices
`let` $H = \max_i\{y_i + h_i\}$ `be the total height of the chip;`
InsertNode $([0, H], 0, black)$; // create the root of the red–black tree
for `each cell` B_i (visited in the order of the topological sort)
  UpdateRedBlackTree $(root, [y_i, y_i + h_i])$;
    // modify the red–black tree to
end_for
    // model the new right border of the analog block
$W = \max\{v.x\}$, $\forall v \in$ `RedBlackTree;`
    // compute the width W of the placement

The procedures *InsertNode* and *DeleteNode* insert/delete a vertex v from the red–black interval tree, preserving the properties of this tree, which were stated at the beginning of this section. The insertion and deletion techniques take $O(\log n)$ time each and are fully discussed in [51]. In addition, the *InsertNode* procedure calls the constructor of a "red–black" node v having as parameters an interval denoted $v.I$ (its low and high extremes being denoted $\min\{v.I\}$ and $\max\{v.I\}$), an abscissa $v.x$ for the computation of the x_i values, and the node color (*red* or *black*). The values of $v.x$ represent abscissae of vertical segments on the right border of the chip.

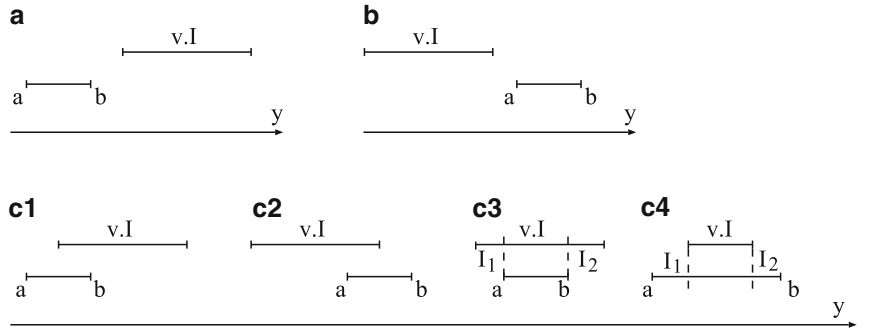


Fig. 1.6 The interval trichotomy for the two – possibly overlapping – closed intervals $v.I$ and $[a, b]$

The procedure *UpdateRedBlackTree* follows the cases of the interval trichotomy of the two intervals $v.I$ and $[a, b]$, that is, the three cases:

(a) $\max[a, b] \leq \min\{v.I\}$;
(b) $\max\{v.I\} \leq \min[a, b]$;
(c) the interiors of the closed intervals $v.I$ and $[a, b]$ overlap; in this last case, there are four situations shown in Fig. 1.6.

procedure *UpdateRedBlackTree* $(v, [a, b])$ // $[a, b] = [y_i, y_i + h_i]$
  if $b \leq \min\{v.I\}$ **then** *UpdateRedBlackTree*$(v.left, [a, b])$; `return;`
    // case (a)
  if $\max\{v.I\} \leq a$ **then** *UpdateRedBlackTree*$(v.right, [a, b])$; `return;`
    // case (b)
  if $v.x > x_i$ **then** $x_i = v.x$;
    // the interiors of $[a, b]$ and $v.I$ overlap: cases (c1-c4)
  if $a < \min\{v.I\}$ && $b < \max\{v.I\}$ **then** // case (c1)
      $v.I = [b, \max\{v.I\}]$; *UpdateRedBlackTree* $(v.left, [a, b])$; `return;`
  if $\min\{v.I\} < a$ && $\max\{v.I\} < b$ **then** // case (c2)
      $v.I = [\min\{v.I\}, a]$; *UpdateRedBlackTree* $(v.right, [a, b])$; `return;`
  if $\min\{v.I\} \leq a$ && $b \leq \max\{v.I\}$ **then** // case (c3)
      if $(I_1 = [\min\{v.I\}, a]) \neq \emptyset$ **then** *InsertNode*$(I_1, v.x, red)$;
      if $(I_2 = [b, \max\{v.I\}]) \neq \emptyset$ **then** *InsertNode*$(I_2, v.x, red)$;
      $v.I = [a, b]$; $v.x = x_i + w_i$;
      MergeAdjacentIntervalsWithSameAbscissae(v); `return;`
  if $a \leq \min\{v.I\}$ && $\max\{v.I\} \leq b$ **then** // case (c4)
      if $(I_1 = [a, \min\{v.I\}]) \neq \emptyset$ **then** *DeleteInterval*$(v.left, I_1)$;
      if $(I_2 = [\max\{v.I\}, b]) \neq \emptyset$ **then** *DeleteInterval*$(v.right, I_2)$;
      $v.I = [a, b]$; $v.x = x_i + w_i$;
      MergeAdjacentIntervalsWithSameAbscissae(v);
end_procedure

In the cases (a) and (b), the procedure *UpdateRedBlackTree* is recursively called for the left and, respectively, right subtree. The cases (c1) and (c2) are similarly handled; the only difference is that the interval $v.I$ is shortened by eliminating the overlap with $[a, b]$ since the intervals in the tree must be disjoint. The number of nodes in the interval tree can increase only in the situation (c3) due to the fragmentation of the interval $v.I$ in at most three segments. On the other hand, the number of nodes in the interval tree can decrease only in the case (c4), when all the nodes (but one) having intervals completely overlapped by $[a, b]$ will be recursively deleted by the procedure *DeleteInterval*, shown below.

The procedure *MergeAdjacentIntervalsWithSameAbscissae* identifies the nodes v_1, v_2 having the intervals adjacent to $v.I$. If v and v_1 and/or v_2 have the same abscissae, the interval of the ancestor is enlarged and the descendent node is removed. For instance, if v is the root in Fig. 1.5 ($v.I = [30, 40]$), v_1 is the node whose interval is [20, 30] (i.e., left of [30, 40]) and v_2 is the node whose interval is [40, 50] (i.e., right of [30, 40]). If, e.g., $v_1.x = v.x$, then the root would get its interval modified to [20,40], while v_1 would be removed, being no longer necessary (since the two segments of the contour would be collinear). Finding the successor and predecessor nodes in a binary search tree is easy [51]. Note that a restoration after deletion of the red–black tree is necessary in this situation; the worst-case complexity of this procedure is $O(\log n)$ [51].

The procedure *DeleteInterval* eliminates (using *DeleteNode*) the nodes with intervals entirely overlapped by $[a, b]$. It is basically working according to the interval trichotomy (see Fig. 1.6) as well.

procedure *DeleteInterval* $(v, [a, b])$
  if $b \le \min\{v.I\}$ **then** *DeleteInterval* $(v.left, [a, b])$; `return;` // case (a)
  if $\max\{v.I\} \le a$ **then** *DeleteInterval* $(v.right, [a, b])$; `return;` // case (b)
  if $v.x > x_i$ **then** $x_i = v.x$;
  if $a \le \min\{v.I\}$ && $b < \max\{v.I\}$ **then** // case (c1)
    if $(I_1 = [a, \min\{v.I\}]) \neq \emptyset$ **then** *DeleteInterval*$(v.left, I_1)$;
    $v.I = [b, \max\{v.I\}]$; `return;`
  if $\min\{v.I\} < a$ && $\max\{v.I\} \le b$ **then** // case (c2)
    if $(I_2 = [\max\{v.I\}, b]) \neq \emptyset$ **then** *DeleteInterval*$(v.right, I_2)$;
    $v.I = [\min\{v.I\}, a]$;
  else // case (c4): **if** $v.I \subseteq [a, b]$ (case c3 cannot appear)
    if $(I_1 = [a, \min\{v.I\}]) \neq \emptyset$ **then** *DeleteInterval*$(v.left, I_1)$;
    if $(I_2 = [\max\{v.I\}, b]) \neq \emptyset$ **then** *DeleteInterval*$(v.right, I_2)$;
    DeleteNode(v);
end_procedure

The red–black tree can have at most n nodes since there are at most n segments on the border contours determined by the y-spanning intervals $[y_i, y_i + h_i]$, hence the red–black tree has a maximum height of $\lceil 2 \log_2 n \rceil$ [51]. Since the node insertions and deletions take $O(\log n)$, we may be tempted to consider $O(\log n)$ the worst-case time bound per iteration. But this is not always true: when new nodes are inserted in

the red–black interval tree (in case (c3)), it can gain at most two nodes per iteration, whereas when the tree decreases in size (in case (c4)), up to $O(n)$ nodes can be deleted.[5]

However, using the *aggregate method* of *amortized analysis* [51], it can be shown that the amortized time bound is $O(\log n)$ per iteration. Intuitively, the reason is that each node can be deleted at most once for each time it is created. In an amortized analysis, the time required to perform a sequence of operations is averaged over all the operations performed. Amortized bounds are weaker though than the corresponding worst-case bounds because there is no guarantee for any single operation. If an average is taken over a sequence of operations, the average cost of an operation may be small, even though a single operation may be expensive.

Using the *aggregate method* of amortized analysis [51], we consider an entire sequence of m *UpdateRedBlackTree* operations on the red–black interval tree having initially only one node. Although the case (c4) can be expensive, the sequence of m operations can cost at most $O(m \log n)$, since each node can be deleted at most once for each time it is created. The amortized cost of an operation is $O(m \log n)/m = O(\log n)$. Since in the algorithm computing the device abscissae there are n iterations, the overall time complexity is $O(n \log n)$. The space requirement of the algorithm is $O(n)$, since the red–black interval tree can have at most $2n - 1$ nodes (the number of vertical segments of the right border contour being at most $2n - 1$).

Example. Consider a layout with ten rectangular blocks, having the widths and heights indicated between parentheses: A(14 × 3), B(3 × 1), C(4 × 2), D(5 × 2), E(4 × 3), F(2 × 6), G(2 × 6), H(2 × 3), I(5 × 6), and J(4 × 2). Assume the cell ordinates are known or have been previously determined: $[y_A \; y_B \ldots y_J] =$ [0 3 4 6 8 3 3 3 6 4]. Let us assume that the order of the nodes in the topological sort of the horizontal constraint graph is alphabetical: A,B, ... ,J. This example illustrates the computation of the device abscissae using a red–black interval tree.

All the block abscissae are initially zero. The first root node v_0 of the red–black interval tree (the first tree in Fig. 1.7) has associated the interval $[0, H] = [0, 12]$ since the height of the chip is $H = \max\{y_i + h_i\} = 12$. The y-spanning interval $[y_A, y_A + h_A] = [0, 3]$ of the first node visited in the topological sort of the horizontal constraint graph is the argument of *UpdateRedBlackTree* in the first iteration. Since in the interval trichotomy the case is as in Fig. 1.6c3, and $I_2 = [3, 12] \neq \emptyset$, the root will get a new right child having attached the interval I_2. The abscissa of block A is $x_A = 0$. The root interval is modified to $v_0.I = [0, 3]$ and the abscissa of the root becomes $v_0.x = x_A + w_A = 14$ (the second tree in Fig. 1.7).

The processing of block B will insert a new node as a red right child in the tree interval (case (b), then case (c3) – Fig. 1.6 – in the recursive call). Since the red node

[5] Such a situation could occur if the red–black tree had $O(n)$ nodes and in the next iteration the block had the height of the whole chip; the red–black tree would be reduced to a single node with $v.I = [0, H]$.

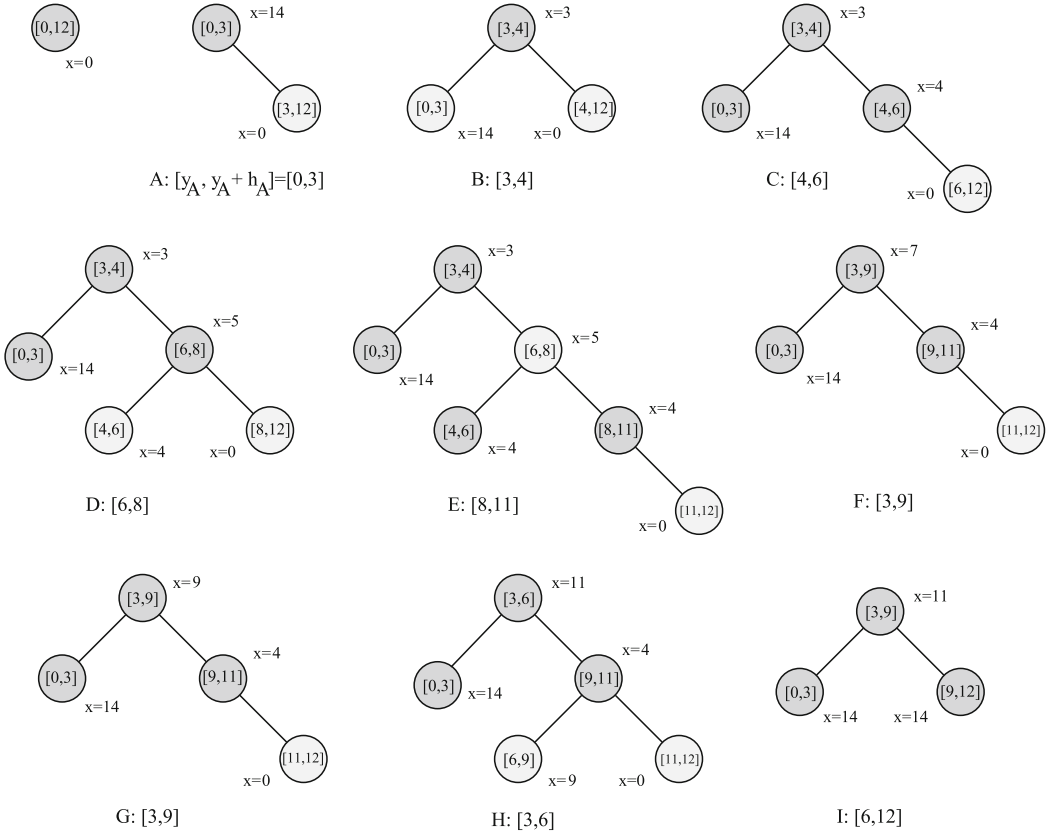


Fig. 1.7 The red–black interval tree evolution. Each node v has attached an interval $v.I$ and an abscissa $v.x$

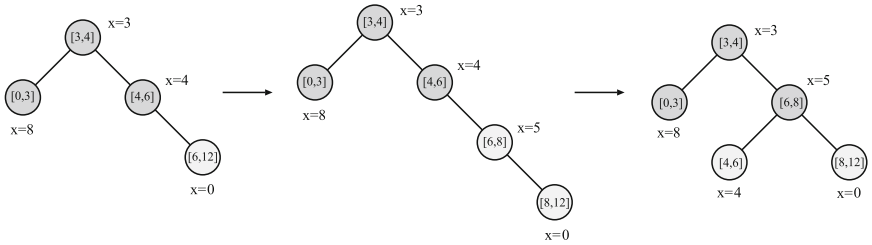


Fig. 1.8 The detailed modification of the red–black interval tree for $[y_D, y_D + h_D] = [6, 8]$

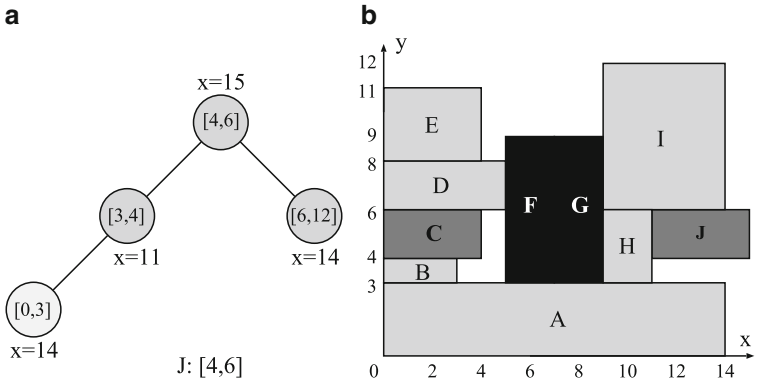


Fig. 1.9 Illustrative example: (**a**) the last red–black interval tree (after the insertion of the y-interval of cell J), and (**b**) the final device placement

[3, 4] has a red child [4, 12] (red–black property violation!), a left rotation as well as a modification of the node coloring are performed to restore the red–black property, the node having the interval [3, 4] becoming the root (the third tree in Fig. 1.7). The operations implied by the processing of block D are displayed in Fig. 1.8: first, a new node in the interval tree is created, followed by a rotation and a change of colors as described in [51].

The successive modifications of the red–black interval tree are displayed in Fig. 1.7. After each iteration, the inorder walk of the red–black interval tree describes exactly the contour of the *right* border of the chip. The computation of the abscissae of the blocks yields: $[x_A\ x_B \ldots x_J] = [0\ 0\ 0\ 0\ 0\ 5\ 7\ 9\ 9\ 11]$. Note that in the "block F"-iteration the case is as in Fig. 1.6c4, since $[y_F, y_F + h_F] = [3, 9]$ covers the intervals [3, 4], [4, 6], and [6, 8] in the red–black tree. The interval of the first node is modified and the other two corresponding nodes are removed by the procedure *DeleteInterval*. The last tree is displayed in Fig. 1.9a; it corresponds to the placement in Fig. 1.9b. The width of the layout is $W = \max\{v.x\} = 15$. □

The evaluation algorithm could also use fully-balanced (AVL) binary search trees [54] instead of red–black trees to achieve the same time complexity. However, in AVL trees balance is maintained by as many as $\Theta(\log n)$ rotations (for the

asymptotic notations, see [51] Chap. 2) after a node deletion,[6] whereas at most two rotations are necessary to maintain the red–black tree after an insertion, and at most three rotations after a deletion [51]. Red–black trees are only *approximately* balanced, but they achieve the same complexity being more efficient in terms of practical computation effort.

1.2.3 Deterministic Skip Lists

Historically, the *probabilistic skip list* (PSL) was introduced first by Pugh [55] as an alternative to balanced search trees [51]. The main idea in the PSL is that each of its keys (i.e., the information contained in the data structure) is stored in one or more sorted linked lists. All keys are stored in sorted order in the linked list denoted as level 1, and each key in the linked list at level $k (k = 1, 2, \ldots)$ is included with probability $p (0 < p < 1)$ in the linked list at level $k + 1$. A *header* contains the references to the first key in each linked list (see the skip list in Fig. 1.10). The *height* of the data structure, that is, the number of linked lists, is also stored.

A search for a key begins at the header of the highest numbered linked list. This linked list is scanned until it is observed that its next key is greater than or equal to the one sought, or the reference is NULL. At that point, the search continues one level below until it terminates at level 1. By convention, the equality test is done only at level 1 as the last comparison; this avoids two tests (or a three-way branch) at each step.

Insertions and deletions are quite straightforward [55]. A new key is inserted when a search for it ends at level 1. As it is put in the linked list $k (k = 1, 2, \ldots)$, it is inserted with probability p when its search terminates at level $k + 1$. This continues until, with probability 1-p, the choice is not to insert. The counter for the height of the data structure is increased, if necessary.

Deletions are completely analogous to insertions. A key to be deleted is removed from the linked lists in which it is found. The height of the data structure is updated by scanning the header's pointers and decreasing the height until a non-NULL pointer (or reference) is found.

PSLs maintain an average logarithmic search and update cost, even after a long sequence of updates. This is in sharp contrast with binary search trees, where it was shown that usual update algorithms lead to degeneration in behavior [56].

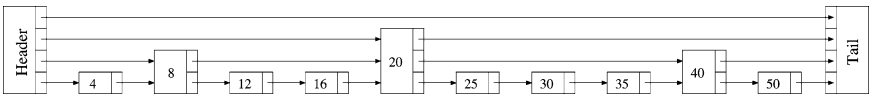


Fig. 1.10 A skip list having the gaps of sizes 1, 2, and 3

[6] The deletion of the leftmost node in a Fibonacci tree [54] is such an example. Rebalancing after insertion never needs more than a single or a double rotation though.

It can be shown that the PSL exhibits an average logarithmic behavior. For instance, consider the search cost for a hypothetical very large key denoted $+\infty$. Clearly, all the n keys are at level 1, about pn keys will make it to level 2, about p^2n keys will make it to level 3, and so on. Therefore, the expected height of the PSL is approximately $\log_{\frac{1}{p}} n$. Since, among all keys that made it to a certain level, about every $\frac{1}{p}$th key will make it to the next higher level, one should expect to make $\frac{1}{p}$ key comparisons per level. Therefore, one should expect $\frac{1}{p}\log_{\frac{1}{p}} n$ in total when searching for $+\infty$.

Despite the fact that *on the average* the PSL performs reasonably well ($\Theta(\log n)$ time for a search or an update [55]), in *the worst case* its $\Theta(n \log n)$ space and $\Theta(n)$ time complexity are considered rather high. In addition, the good *average case* performance of the PSL, although independent of the input, does depend on the random number generator behaving "as expected." Should this not be the case at a particular instance (if, for instance, the random number generator creates elements (nodes) of equal heights[7]), the PSL may degenerate into a structure worse than a linear linked list. On the other hand, a class of *deterministic skip list* (DSL) [57] achieves logarithmic worst-case costs.

Assuming that a skip list of n keys has a 0th (`Header`) and a $(n+1)$st node (`Tail`) of height equal to the height of the skip list, two elements are *linked* when there exists at least one pointer going from one to the other. Given two linked elements, one of the height exactly $h (h > 1)$ and another of height h or higher, their gap size is the number of nodes of height $h-1$ that exist between them. For instance, in the skip list of Fig. 1.10 the gap size between 20 and 40 is 3, while the gap between $-\infty$ and $+\infty$ (`Header` and `Tail`) is 1. A *deterministic skip list* is a skip list having the gap sizes in a given range.

It can be proven [58] that there is a one-to-one mapping between the set of B-trees of order m [54], $m \geq 3$, and the set of DSLs with gaps of sizes $\lceil \frac{m}{2} \rceil - 1$, $\lceil \frac{m}{2} \rceil$, ..., $m-2$, or $m-1$. Consequently, this class of DSLs achieves logarithmic worst-case complexities for search and update (insert/remove key) [57]. When $m = 2k+2$, this subclass of DSLs having gap sizes between k and $2k+1$, called $k-(2k+1)$ DSLs has an additional desirable property: the insertion and deletion can be implemented using a top-down strategy in a relatively easy way. The simplest DSL of this subclass ($k = 1$), the so-called 1-3 (or 1-2-3) DSL having only gaps of size 1, 2, or 3 (like the one in Fig. 1.10), may be the main data structure used for the computation of the layout contour in some evaluation algorithm.

1.2.3.1 Insertion and Deletion of a Key in a 1-3 DSL

Adopting a top-down approach, we choose to perform an insertion in a 1-3 DSL by splitting any gap of size 3 on our way to the bottom level into two gaps of size 1. We ensure in this way that the data structure retains the gap invariant with or without

[7] The height of an element is the number of linked lists to which it belongs. The height of a skip list is the maximum height of its list nodes.

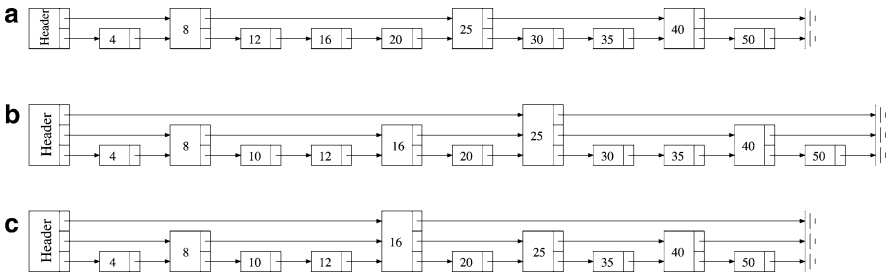


Fig. 1.11 (**a**) A 1–3 skip list. (**b**) The skip list after the top-down insertion of the key 10. (**c**) The skip list after the top-down deletion of the key 30

the inserted key. To be more precise, the search starts at the header, one level higher than the height of the skip list. If the gap we are going to drop in has the size 1 or 2, then we simply drop. If the gap is of size 3, first we raise the middle element in the gap, creating thus two gaps of size 1 each, and only then we drop. When the bottom level is reached, we simply insert a new element (node) of height 1. Since this algorithm allowed only gaps of sizes 1 and 2 before the proper insertion, the newly inserted element does not modify the gap invariant and, therefore, leaves a valid 1-3 DSL.

As an example, consider the case of inserting 10 in the skip list of Fig. 1.11a. We start at level 3 of the header, we look at level 2 where 25 is raised, then we drop at level 2. We move to level 2 of node 8, we look at level 1 and we raise the node 16. Then we drop to level 1 of 8, and finally we insert 10 as a new node of height 1. The resulting skip list is shown in Fig. 1.11b.

To delete a key from a 1-3 DSL, we work in a top-down manner as well. The search preceding the actual key removal should have the side effect to leave each gap legal, but above the minimum size of 1 as we pass through it. This is handled by either merging with a neighbor, or borrowing from a neighbor. More precisely, the search is started at the header and at the level equal to the height of the skip list. If the gap G that we are going to drop in is of size 2 or 3, then we simply drop. If the gap G is of size 1, we proceed as follows. If G is not the last gap on the current level, then if the following gap G' is of size 1, the gaps G and G' are "merged" by lowering the element separating the two gaps. If the following gap G' is of size 2 or 3, then we "borrow" from it: the node separating G and G' is lowered, whereas the first element of G' is raised. On the other hand, if G is the last gap on the current level, then we "merge" with or borrow from its preceding gap. We continue in this way until the bottom level 1 is reached. There, we remove the key if its node has the height equal to 1. Otherwise, the node is swapped with its predecessor, followed by the removal. Since this algorithm does not allow any gaps to be of size 1, what we are left with after the removal of the element of height 1 is a valid 1-3 DSL.

As an example, consider the deletion of the key 30 in the skip list of Fig. 1.11b. We start at level 3 of the header, we move to level 3 of node 25. We look at level 2

and, since the gap has size 1 and it is the last one on the current level, we look at the preceding gap whose size is 2. Then we "borrow" from the preceding gap, lowering the node 25 separating the two gaps, while raising node 16 – the nearest element from 25 in the preceding gap. Then we drop at level 2 and we look at level 1; we drop at level 1 of 25, and finally remove 30. The resulting skip list is shown in Fig. 1.11c.

This top-down insertion and deletion approaches are easily generalizable to any $k-(2k+1)$ skip list, for $k = 1, 2, \ldots$. When inserting a key in a 2-5, 3-7, 4-9, ... DSL, we split a gap of size 5, 7, 9, ... into two gaps of legal size 2, 3, 4, ... before we drop down a level. When deleting a key from such a DSL, we merge/borrow if we are going to drop in a gap of size 2, 3, 4, ...

1.2.3.2 Implementation Aspects of 1-3 DSL's

The 1-3 DSL achieves logarithmic worst-case search and update complexities if its elements are implemented either as linked lists, or as arrays of exponential heights (the so-called *horizontal array implementation*) [58]. Since the memory requirements for this placement algorithm do not impose severe constraints,[8] we adopted the *linked list implementation* as it is credited with more clarity, elegance, and simplicity.

The implementation is somewhat tricky. The nodes do not contain arrays of forward links as it would seem natural: with such a strategy, promoting a height h node to height $h+1$, $O(h)$ time is needed only to copy the h links, and the time bound would result $O(\log^2 n)$ per insertion/deletion. Instead, favoring time versus space, each `DslNode` in the implementation maintains a link `down` to descend a level, a link `right` to the next node on the same level, and the key that is logically stored in the *next* DSL element. The actual implementation of the DSL in Fig. 1.10 is shown in Fig. 1.12. Notice that some keys appear more than once: if a node has height h in the DSL, its key will appear in h places in the actual implementation. To make the code faster and simpler (avoiding having special cases in the code), a dummy `head` node, and two sentinel nodes `bottom` and `tail` are used to replace the NULL links downward and, respectively, to the right.

An implementation (in C++ style) of the procedure inserting a key [58] is given below. The data members of a `DslNode` are the key, and two pointers – one downward and one to the right.

```
void DSL::insert (int KEY)
{   DslNode *p = head, *t, *pdr, *pdrr;
    bottom->key = KEY;
    while ( p!=bottom )
    {  while ( p->key < KEY ) p = p->right;
       if ( p->key > (pdrr=(pdr=p->down->right)->right)->key )
```

[8] Analog circuits seldom contain more than 100 cells per hierarchical level [2].

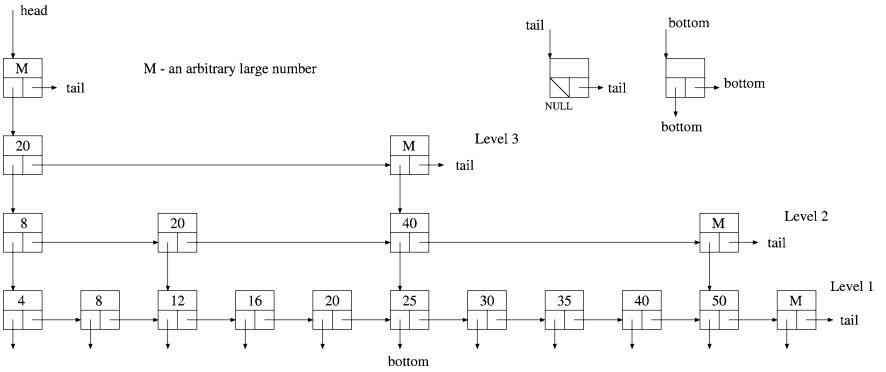


Fig. 1.12 Linked list implementation of the 1-3 DSL in Fig. 1.10. Note the presence of several `DslNode`'s having the same key for each element (node) in the abstract representation of the DSL in Fig. 1.10. The sentinel nodes *tail* and *bottom* are also shown

```
        {  p->right = new DslNode(p->key,pdrr,p->right);
           p->key = pdr->key; p->x = pdr->x;
        }
        else
          if ( p->down == bottom ) return; // KEY already in the DSL
        p = p->down;
    }
    if ( head->right != tail ) head = new DslNode(M,head,tail);
};
```

Adopting a top-down approach, the *insert* procedure ensures that the DSL retains the gap invariant with or without the inserted key by splitting any gap of size 3 on the way from the `head` to the bottom level into two gaps of size 1. To be more precise, we start the search at the `head`, and if the gap we are going to drop in is of size 1 or 2, we simply drop; if the gap is of size 3, first the middle element in this gap is raised one level, creating thus two gaps of size 1 each, and then we drop. When the bottom level is reached, a new element of height 1 is inserted, and the new DSL remains valid (therefore, it is still a 1-3 DSL).

1.2.3.3 Algorithm Computing the Border Contour of the Layout

In this section, we are going to use a 1-3 DSL data structure to compute the device abscissae x_i assuming the device ordinates y_i are already known. Similarly as in the previous sections, it is assumed that a topological sort of the horizontal constraint graph is available (derived from the topological representation encoding the layout).

The deterministic skip list is used here to register the fragmentation of the right or left contour of the (partial) placement configuration. The keys of the DSL are the y-coordinates where the contour of the right border of the analog block is broken. Therefore, the maximum key is H – the height of the analog block (which is already

known since the y-coordinates have been already computed) – and the minimum key is zero. Each element (node) n in the DSL has attached the key $n.key$ and a value $n.x$, which is the abscissa of the border segment starting from the ordinate $n.key$ upward.

The devices are visited in the order of the topological sort, such that the blocks to the *left* are visited before the ones to the *right*, such that the horizontal positioning constraints (induced by the topological representation) be satisfied. When the node B_i is visited, the DSL is updated storing the contour of the *right* border as a result of the insertion of the new y-spanning interval $[y_i, y_i + h_i]$.

Algorithm: Computation of the device abscissae (x_i) using a 1-3 DSL

`let` $x_i = 0$`;` // reset all the abscissae of the left-bottom corners of the devices
`DSL.insert(0);` // the key 0 is inserted as a sentinel in the DSL
for `each cell` B_i (visited in the order of the topological sort)
UpdateDSL $([y_i, y_i + h_i])$`;` // modify the DSL to model the right border
end_for // of the analog block; x_i are updated as well
$W = \max\{n.x\}$, $\forall n \in$ `DSL;` // compute the width W of the placement

The procedure *UpdateDSL*, shown below, modifies the DSL that stores the lateral contour when a new device B_i is added to the partial placement as shown in Fig. 1.13a. To modify the contour of the right border due to the block B_i, the largest key (y-coordinate) $\leq a$ (y_j in Fig. 1.13a) is detected first. All the larger keys inside the interval (a, b) (that is, y_{j+1} to y_k in Fig. 1.13a) must be removed from the DSL and new keys a and b are inserted. (Special care is taken when a and/or b coincide with y_j, respectively, y_k.) x_i – the abscissa of the left-bottom corner of B_i, is the maximum of the x-coordinates attached to the DSL nodes having the keys $y_j, y_{j+1}, \ldots, y_k$. To keep minimal the number of nodes in the DSL, the neighbors of the node having the key a are removed if they have attached the same x-coordinates (since the segment AB in Fig. 1.13a would be collinear with the neighbor segments of the contour).

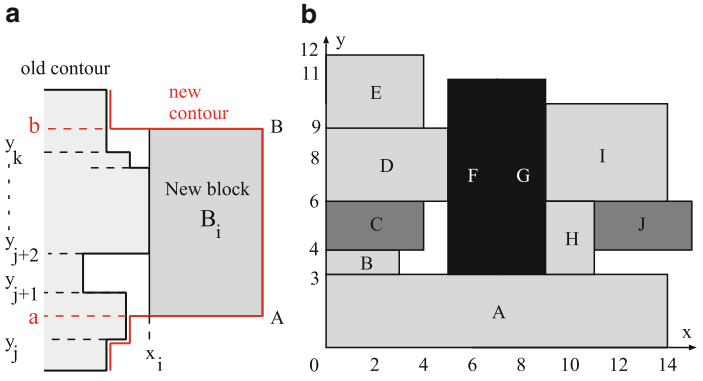


Fig. 1.13 (**a**) Modification of the border contour in the procedure `UpdateDSL` (`[`y_i, $y_i + h_i$`]`) when a new block B_i is processed. (**b**) Final device placement corresponding to the last 1-3 DSL in Fig. 1.14

```
procedure UpdateDSL ([a,b])  // [a,b] = [y_i, y_i + h_i]
  let q be the DslNode in the DSL whose key
  is the largest ≤ a;
        // that is, the node with the key y_j as in Fig. 1.13a
  x_i = max{p.x},  ∀ DslNodes p such that p.key∈[q.key,b)
  if (b is not a key in the DSL)
  {  DSL.insert(b);
        // insert new key b in the DSL if this key does not exist
     DslNode(b).x = predecessor(DslNode(b)).x;
        // ... and set the abscissa of its DslNode the same as its predecessor's one
  }     // (that is, the DslNode with the key y_k as in Fig. 1.13a)
  if (q.key < a)  DSL.insert(a);
        // insert new key a in the DSL if it does not exist
  for all the nodes p such that p.key∈(a,b)
     DSL.remove(p.key);    // remove the keys covered by (a,b):
  end_for                  // that is, y_{j+1} to y_k in Fig. 1.13a
  DslNode(a).x = x_i + w_i;
        // to keep the DSL size minimal, the adjacent collinear contour segments
        // ... are merged by removing the keys with identical abscissae
  if ( DslNode(a).x == DslNode(b).x )  DSL.remove(b);
  if ( a > 0 && predecessor(DslNode(a)).x == DslNode(a).x)
     DSL.remove(a);
end_procedure
```

Example. Consider a layout with ten rectangular blocks: A(14 × 3), B(3 × 1), C(4 × 2), D(5 × 3), E(4 × 3), F(2 × 8), G(2 × 8), H(2 × 3), I(5 × 5), and J(4 × 2) – where the widths and heights are indicated between parentheses. Assume the cell ordinates are known or have been previously determined: $[y_A\ y_B \ldots y_J] =$ [0 3 4 6 9 3 3 3 6 4]. Let us assume that the order of the nodes in the topological sort of the horizontal constraint graph is alphabetical.

The computation of the device abscissae is initiated by inserting the key 0 as a sentinel into the empty DSL. In this way, the procedure finding the node whose key is lesser than a certain positive value (see the pseudocode of *UpdateDSL*) will never fail since all the keys $y_i \geq 0$.

The first visited node in the binary tree is A; since the y-spanning interval of device A is $[y_A, y_A + h_A] = [0, 3]$, the keys 0 and 3 must be inserted in the DSL provided they do not exist already, while removing the keys covered by the interval [0, 3]. Only the key 3 needs to be inserted and no other key is deleted. The procedure *UpdateDSL* with the yields $x_A = 0$ and assigns to the node 0 the x-value $x_A + w_A = 14$ (see Fig. 1.14). The visit of the subsequent nodes B, C, D, and E modifies the DSL in a similar way.

Since the y-spanning interval of the device F is $[y_F, y_F + h_F] = [3, 11]$, the keys 4, 6, and 9 – covered by this interval – are removed, and the key 11 is inserted into the DSL (see the 7th DSL in Fig. 1.14). The abscissa x_F is the maximum of the x-values of the node 3 and of the removed nodes (4, 6, and 9), that is,

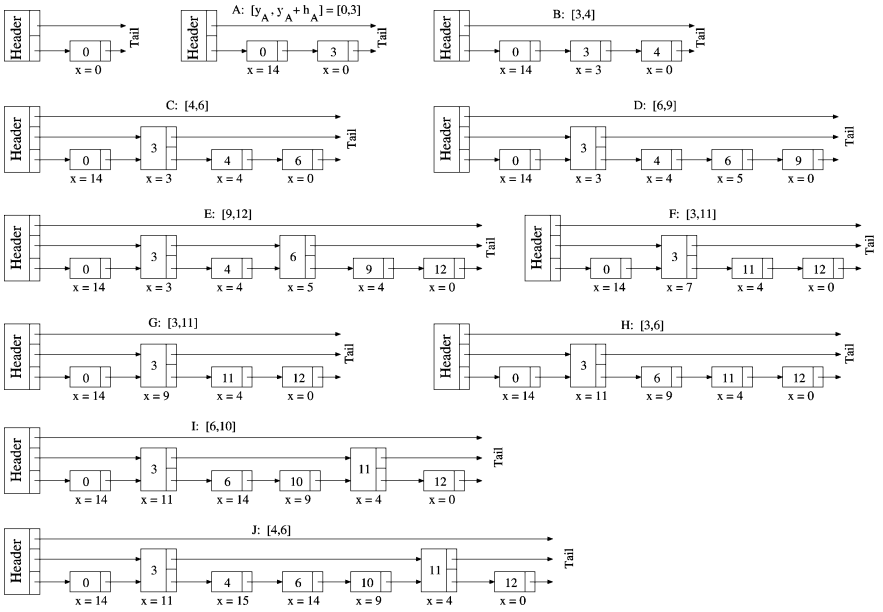


Fig. 1.14 The 1-3 DSL during the computation of the device abscissae. Each list node has attached the x-coordinate of a segment of the right contour starting at the y-coordinate equal to the key of the node. The last skip list corresponds to the layout in Fig. 1.13b

$x_F = \max\{3, 4, 5, 4\} = 5$. The x-value of the node 3 becomes $x_F + w_F = 7$, while the x-value of the node 11 inherits the abscissa of node 9 (the last node removed), that is, 4.

The last DSL in Fig. 1.14 corresponds to the placement in Fig. 1.13b. Note that after each iteration, the DSL models the contour of the right border of the partial placement, the keys being the y-coordinates where the contour changes direction. □

The 1-3 DSL can have at most $n + 1$ elements as there are at most n segments on the border contours determined by the y-spanning intervals $[y_i, y_i + h_i]$. Since the key insertions and deletions take $\Theta(\log n)$ [58], we may be tempted to consider $O(\log n)$ the worst-case time bound per iteration. But this is not always true: when the DSL grows, it can gain at most two new elements per iteration, but when it decreases in size (see the visit of node F in Fig. 1.14), as many as $O(n)$ keys can be deleted.[9] Using the *aggregate method* of *amortized analysis* [51], it can be shown that the amortized time bound is $O(\log n)$ per iteration. Intuitively, the reason is that each key can be deleted at most once for each time it is inserted. In an amortized

[9] Such a situation could occur if the DSL had $O(n)$ elements and in the next iteration the block had the height of the whole chip; the DSL would be reduced to only 2 nodes having the keys 0 and H.

analysis, the time required to perform a sequence of operations is averaged over all the operations performed. Since there are n iterations – one for each device, the overall time complexity is $O(n \log n)$.

The space requirement of the algorithm is $O(n)$, since on each level k of the DSL implementation (see Fig. 1.12a) there are at most $\lfloor \frac{n+2}{2^{k-1}} \rfloor$ `DslNode`'s, therefore, at most $2n + 7$ in total, taking also into account the nodes `header`, `tail`, and `bottom`.

1.2.4 *Johnson's Priority Queue*

The keys, integers in the set $\{1, \ldots, N\}$, in Johnson's priority queue model [32] are kept in N *buckets*. The nonempty buckets, together with bucket 0 (always present and used as a header) are kept in the *bucket list*, a doubly-linked list sorted on the key values. The buckets in the list correspond to the leaves of a binary tree T. The nodes of T are indexed by defining a complete [51] *host* binary tree $H = \{1, \ldots, 2^h + N\}$, where $h = \lceil \log_2(N + 1) \rceil$. Each node q in H, different from the root, is a child of the node $\lfloor q/2 \rfloor$.

Example. Figure 1.15 shows a priority queue for keys in the set $\{1, \ldots, 6\}$, containing, in addition to bucket 0, only the keys 2 and 4. The list of buckets is shown below the leaves of the tree T, drawn with bold solid lines. The host tree H is represented with dashed lines. Because of reason of space, the figures in the rest of the chapter will show only the bucket lists of the priority queue. □

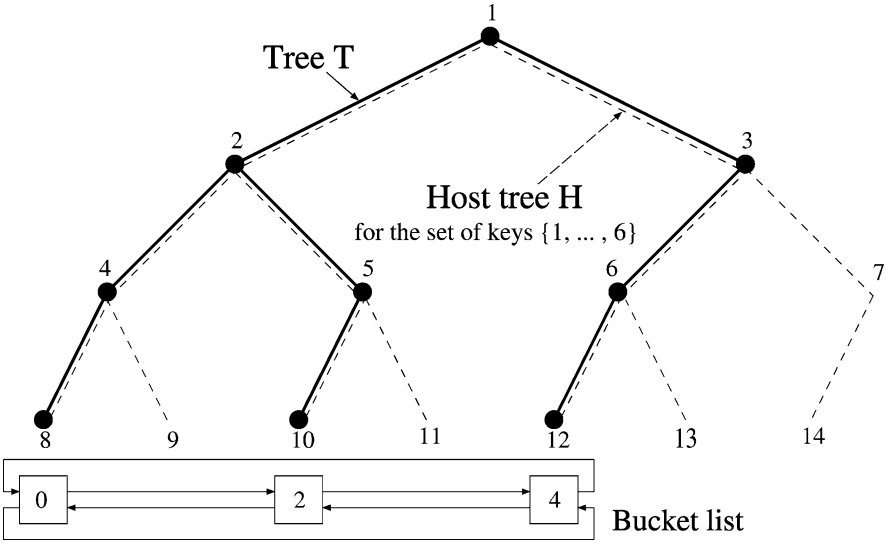


Fig. 1.15 Johnson's priority queue [32]

The driving idea is to conceive a complete binary tree with $N + 1$ leaves on the lowest level, but dynamically to leave much of the tree incomplete. Whenever a new bucket is inserted, its path would be constructed upward, until the new path intersected the path of some nonempty bucket. Then, the path would be followed downward toward leaves to find the nonempty bucket adjacent to which the new bucket must be inserted into the list. Deletion would be a reversal of this process.

Since the height h of the host binary tree is $O(\log N)$, and a path in the tree T of length k is traversed using a mechanism somewhat similar to the binary search, visiting at most $2\lceil \log_2 k \rceil$ nodes [32], the key insertions and deletions take $O(\log \log N)$ time.

An algorithm computing the devices' abscissae will look very similar to the one described in Sect. 1.2.3, where the deterministic skip list is replaced by a priority queue with its list of buckets. The advantage is that updating the priority queue would have an amortized time bound of $O(\log \log n)$, yielding an overall time complexity of $O(n \log \log n)$, hence better than using a DSL.

1.3 Symmetric-Feasible Sequence-Pairs

Dealing efficiently with symmetry constraints in the framework of topological representations implies addressing two problems:

a. How to recognize encodings complying with the given symmetry constraints, without building the corresponding layout, and
b. How to restrict the exploration of the solution space of the representation only to "symmetric-feasible" (S-F) codes.

This section will address the questions above assuming *sequence-pair* [29] as the topological representation. The basic idea of the sequence-pair encoding, briefly described below for the sake of consistency, is to represent any rectangle packing as an ordered pair of cell sequences (α, β). Denoting by α_i, the cell of index i (occupying the position i) in sequence α, and by α_A^{-1} the position of the cell A in the sequence,[10] the topological relations between two cells A and B are given by the following rules:

If $\alpha_A^{-1} < \alpha_B^{-1}$ and $\beta_A^{-1} < \beta_B^{-1}$ then cell A is to the left of cell B;
If $\alpha_A^{-1} < \alpha_B^{-1}$ and $\beta_B^{-1} < \beta_A^{-1}$ then cell A is above cell B.

For instance, a sequence-pair encoding of the 7-cell placement configuration in Fig. 1.16a is $(\alpha, \beta) = (CDAFBGE, DCBGAFE)$. With the notations employed, we have, e.g., $\alpha_1 = C$, $\beta_4 = G$, and also, $\alpha_C^{-1} = 1$, $\beta_C^{-1} = 2$, etc. As $\alpha_F^{-1} < alpha_B^{-1}$ and $\beta_B^{-1} < \beta_F^{-1}$ $(4 < 5$ and $3 < 6)$, it follows that cell F is positioned above cell B.

[10] Since α and β are one-to-one mappings, the inverse functions α^{-1} and β^{-1} are well defined.

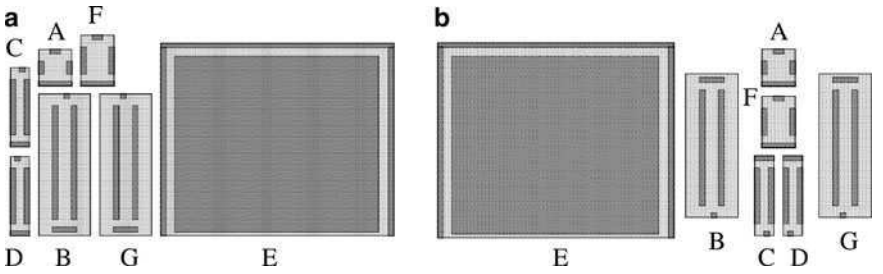


Fig. 1.16 (**a**) Placement configuration encoded by the sequence-pair (*CDAFBGE*, *DCBGAFE*). (**b**) Placement with symmetry group {(C,D), (B,G), A, F} corresponding to the S-F sequence-pair (*EBAFCDG*, *EBCDFAG*)

Now, let (α, β) be the sequence-pair of a placement configuration containing a number of symmetry groups (each group composed of pairs of symmetric devices and self-symmetric devices relative to a common vertical axis). Denoting $\mathrm{sym}(x)$ the symmetric pair of cell x, the sequence-pair (α, β) is called *symmetric-feasible* (S-F) [43] if for any distinct cells x, y in any of the symmetry groups

$$\alpha_x^{-1} < \alpha_y^{-1} \iff \beta_{\mathrm{sym}(y)}^{-1} < \beta_{\mathrm{sym}(x)}^{-1} \tag{1.1}$$

Taking $y = \mathrm{sym}(x)$, and noting also that $\mathrm{sym}(\mathrm{sym}(x)) = x$, the condition (1.1) shows that any symmetric pair of cells appears in the same order in both sequences α and β. In addition, two cells x, y belonging to distinct symmetric pairs appear in one of the sequences in reversed order as do their symmetric cells in the other sequence. Note that the condition (1.1) works neatly also when self-symmetric cells (having $x = \mathrm{sym}(x)$) are involved.

Example. Assuming that a given subset of the placeable cells must constitute a symmetry group, not all the sequence-pair codes are feasible any more. For instance, suppose the pair of cells (C, D) in Fig. 1.16a should be symmetric relative to a vertical axis: the encoding $(\alpha, \beta) = (CDAFBGE, DCBGAFE)$ is not feasible as it leads to a placement configuration where cell C lays above D.

Figure 1.16b displays a placement corresponding to the S-F sequence-pair (*EBAFCDG*, *EBCDFAG*). Assuming a symmetry group $\{(C, D), (B, G), A, F\}$ composed of two symmetric pairs and two self-symmetric cells A and F, the sequence-pair above is symmetric-feasible. Indeed, taking the self-symmetric cell A and comparing its positions $\alpha_A^{-1} = 3$ and $\beta_A^{-1} = 6$ in the sequences α and β to the corresponding positions of the other cells in the group, it follows that the condition (1.1) is satisfied whenever cell A is involved. Similar comparisons involving the positions of the other cells in the symmetry group will conclude the verification. □

Important remark: The property (1.1) is a *sufficiency* condition: it ensures the building of a valid placement in symmetry point of view, as it will be shown in Sect. 1.3.1. Intuitively, the property (1.1) prohibits the situation when two cells from distinct

symmetric pairs are in an "above-below" topological relation, while their pairs are in the reverse relation, the two pairs preventing each other to align horizontally. It also prohibits the situation when two symmetric pairs are intertwined, preventing each other to align vertically about the same symmetry axis. Since the sequence-pair typically presents redundancies, one can find sequence-pairs whose evaluation yields a valid placement in symmetry point of view, but still do not satisfy the property (1.1). We called the sequence-pairs having property (1.1) *symmetric-feasible* since they can certainly generate symmetrically valid placements (see Sect. 1.3.1). But it must not be construed that any sequence-pair not satisfying property (1.1) is automatically symmetric-unfeasible. For instance, [21] shows a placement example satisfying given symmetry constraints whose unique sequence-pair does not satisfy the property (1.1). Such examples are rare though and they may be dependent on the dimensions of the cells. The benefit of the sufficient condition (1.1) is that the exploration of the solution space of placement problems with symmetry constraints can be reduced to the exploration of those sequence-pairs, which are *symmetric-feasible*, i.e., satisfy property (1.1) relative to every symmetry group of cells. The positive outcome is a significant reduction in size of the search space. The magnitude of this reduction is given by the following

Lemma 1. *The number of symmetric-feasible sequence-pairs corresponding to a placement configuration with n cells and G symmetry groups, each group k containing p_k pairs of symmetric cells and s_k self-symmetric cells ($k = 1, \ldots, G$), is upper-bounded by* $\frac{(n!)^2}{(2p_1+s_1)!\cdot\cdots\cdot(2p_G+s_G)!}$

Proof. There are $C_n^{2p_1+s_1}$ sets of positions in the sequence α (and the same number in the sequence β) for the $2p_1 + s_1$ cells of the first symmetry group. Similarly, there are $C_{n-2p_1-s_1}^{2p_2+s_2}$ sets of positions in any of the two sequences occupied by the $2p_2 + s_2$ cells of the second group, and so on.

Now, there are $(2p_1 + s_1)!$ possibilities of ordering the cells of the first group in the sequence α. Note that for each order of these cells in α, their order in the sequence β is unique due to the condition (1.1) above. The same is true for any of the G symmetry groups. However, the contribution of the cells that do not belong to any of the groups is $[(n - 2\sum_k p_k - \sum_k s_k)!\,]^2$ since their order in the sequence α is independent of their order in β.

In conclusion, the number of S-F sequence-pairs is upper-bounded by

$$\left[C_n^{2p_1+s_1}\cdot C_{n-2p_1-s_1}^{2p_2+s_2}\cdot\ \ldots\right]^2\cdot(2p_1+s_1)!\cdot(2p_2+s_2)!\cdot\ldots\cdot\left[(n-2\sum_k p_k-\sum_k s_k)!\right]^2$$

which yields the result in the *Lemma* after expansion and simplification. □

For instance, the number of S-F sequence-pairs for the example in Fig. 1.16b, where $n = 7$ and $p_1 = s_1 = 2$, is $(7!)^2/6! = 35,280$, whereas the total number of sequence-pairs is $(n!)^2$=25,401,600 (therefore, a reduction of the search space of 99.86%).

1.3.1 Evaluation of Symmetric-Feasible Sequence-Pairs

Given a sequence-pair (α, β), the maximal (i.e., which cannot be enlarged) subsequences common to α and β represent paths in the horizontal constraint graph of (α, β). For instance, $EBCDG$, $EBAG$, and $EBFG$ are common subsequences of the sequence-pair ($EBAFCDG$, $EBCDFAG$) (see Fig. 1.16b). If the cells are weighted with their widths, the weight of the longest common subsequence (LCS) $EBCDG$ represents the width of the block placement [31]. Similarly, the maximal subsequences common to α^R (sequence α reversed) and β represent paths in the vertical constraint graph of (α, β) – as, for instance, CFA (see Fig. 1.17b). If the cells are weighted with their heights, the longest common subsequence represents the height of the whole placement. Based on these concepts, an evaluation algorithm using the priority queue model described in Sect. 1.2.4 that builds the block placement from a given sequence-pair in $O(n \log \log n)$ time was presented in [31].

This section will present an algorithm using the LCS approach building a placement *subject to symmetry constraints* (as explained in the introduction section) from a given *symmetric-feasible* sequence-pair. The analog devices to be placed on the chip area are represented by n rectangular blocks $B_1, \ldots, B_n$, each block B_i having the width w_i and the height h_i, and having (x_i, y_i) as coordinates of its left-bottom corner. The algorithm presented in this section assumes for the time being that all the devices subject to symmetry constraints belong to a single symmetry group. The implementation takes into consideration an arbitrary number of symmetry groups, though. The extension to multiple symmetry groups is addressed in Sect. 1.3.2.

The evaluation algorithm uses a *priority queue* whose model was presented in Sect. 1.2.4. However, it must be emphasized that both the computation of the device ordinates and device abscissae can be slightly modified to work with given topological sorts of the vertical and, respectively, horizontal constraint graphs of the placement. Consequently, these computations can be done using either segment trees (see Sect. 1.2.1), or red–black interval trees (see Sect. 1.2.2), or 1–3 deterministic skip lists (see Sect. 1.2.3).

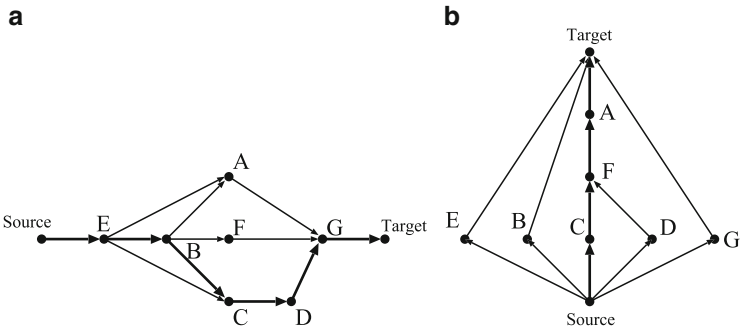


Fig. 1.17 (**a**) The horizontal and (**b**) vertical constraint graphs of the placement from Fig. 1.16b

Similar to [31], each bucket in the priority queue has associated two keys (*index*, *length*), where the *index* key is the cell position in sequence β and *length* is the length of the LCS until that cell in the sequence-pair. Since the number of buckets is at most $n + 1$, the insertion and deletion operations take $O(\log \log n)$ time [32].

1.3.1.1 The Computation of the Device Ordinates

Step 1y: The device ordinates must be determined first. (The reason will become apparent when computing the abscissae.) The algorithm deriving the y-coordinates (initially, zero) of the devices performs the LCS computation with the sequence α in reverse order. The basic difference from [31] is that the equality of symmetric devices' ordinates must be enforced. Also, if a y-coordinate that was previously computed must be subsequently increased due to symmetry, then *Step 1y* must be repeated since, otherwise, some vertical topological constraints may be violated.

```
insert bucket (0,0) in the priority queue;
        // this special bucket acts as a sentinel
for each index i in α (i = n to 1)
  let l be the index in β of cell B_j = α_i ;
  find bucket pred_l whose index is the largest lesser
  than l ;
       // pred_l does always exist due to the sentinel bucket (0,0)
  y_j = max {y_j , length of pred_l};
  if B_j has a symmetric B_k
  then if B_k has already been visited and y_k < y_j
       then Step 1y is repeated; // restarting with the y's obtained
       y_k = y_j ;
  insert bucket (l, y_j + h_j) into the queue;
  remove buckets with an index
         greater than l and a length lesser than or equal
           to y_j + h_j ;
end_for
remove all buckets from the priority queue;
```

Example 1. Let (*EDCKAFGIHJBL*, *KACDEFGLHBJI*) be a symmetric-feasible sequence-pair relative to a symmetry group consisting of three pairs of symmetric devices (F,G), (K,L), and (C,J). The successive modifications of the bucket list during the first four and last three iterations are displayed in Fig. 1.18a–h. The current height of the placement is 11, the *length* field in the last bucket (5,11) in Fig. 1.18h. A second execution of this step is necessary since y_J, initially set to 3 (see Fig. 1.18d), becomes equal to 4 when y_C is computed, increase which creates a topological violation.[11] □

[11] Actually, one may resume from cell J's iteration: instead of (11, 5), the bucket (11, 6) will be inserted in Fig. 1.18d.

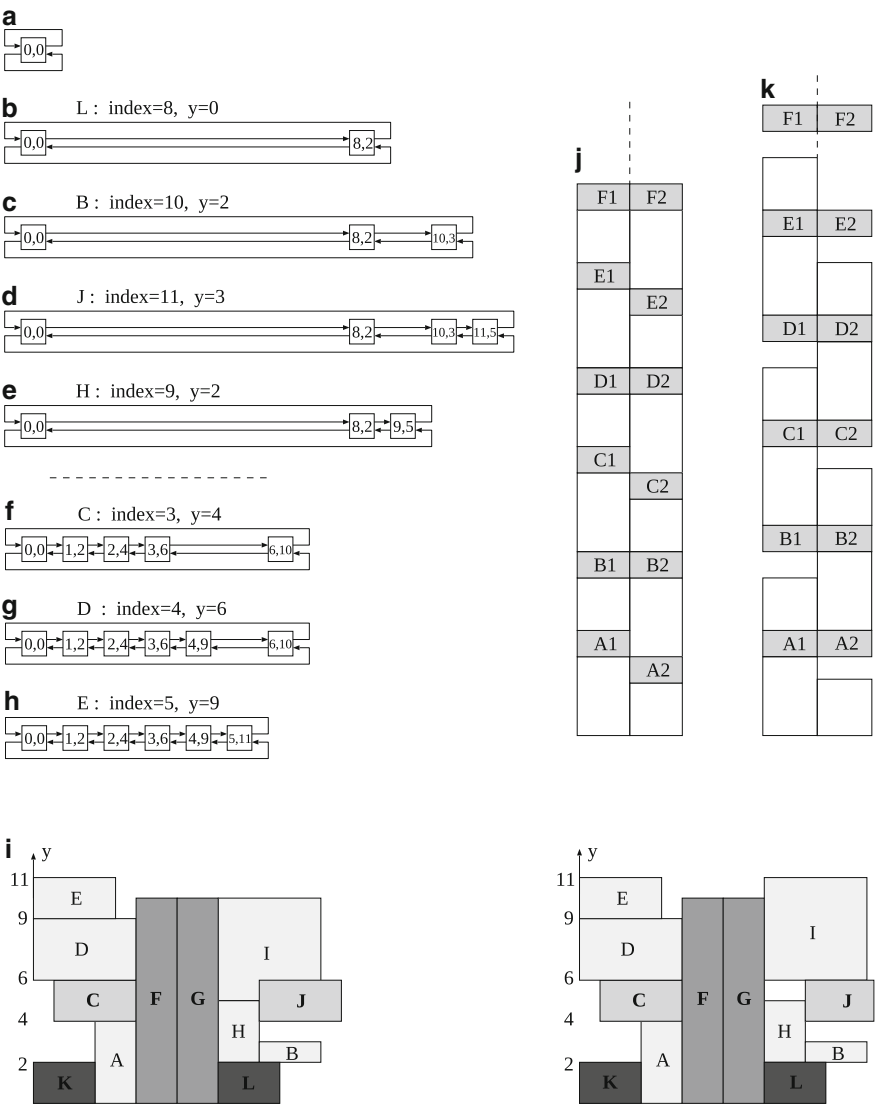


Fig. 1.18 Example 1: the computation of y-coordinates for the symmetric-feasible sequence-pair (*EDCKAFGIHJBL*, *KACDEFGLHBJI*) with a group of three pairs of symmetric devices (F, G), (K, L), and (C, J). The widths and heights of the devices are: $A(2 \times 4)$, $B(3 \times 1)$, $C(4 \times 2)$, $D(5 \times 3)$, $E(4 \times 2)$, $F(2 \times 10)$, $G(2 \times 10)$, $H(2 \times 3)$, $I(5 \times 5)$, $J(4 \times 2)$, $K(3 \times 2)$, and $L(3 \times 2)$. (**a**–**h**) The update of the bucket list of Johnson's priority queue during the first four and last three iterations. The cell processed in the current iteration is mentioned, together with its index in β and its computed ordinate. The positioning along the axis Ox is irrelevant here. (i) Device placements along the Oy axis: (*left*) after the first execution of *Step 1y* (note that cell J overlaps cell I); (*right*) after the second execution of *Step 1y*. (**j**–**k**) Example where the computation of y-coordinates necessitates $\Theta(p)$ iterations, p being the number of symmetric pairs

The "symmetric-feasibility" property (1.1) prohibits the situation when two cells from distinct symmetric pairs are in an "above-below" topological relation, while their pairs are in the reverse relation, the two pairs preventing each other to align horizontally; therefore, it ensures that after a finite number of executions of *Step 1y*, all the vertical topological and symmetry constraints will be satisfied. Moreover, the height of the resulting placement is minimal since the y-coordinates are computed using the longest common subsequence approach, the weights being the heights of the cells, and also since the vertical symmetry constraints are fixed in a bottom-up direction, with a minimum increase of the ordinate of the lowest cell in the symmetric pair.

The priority queue has at most $n + 1$ buckets; the *insert* and *remove* operations in this data structure can be performed in $O(\log \log n)$ time [32]. Although in some iterations $O(n)$ buckets may be discarded, the amortized complexity per iteration is $O(\log \log n)$. Note that *Step 1y* may need to be executed $\Theta(p)$ times, where p is the number of symmetric pairs in the group. Fig. 1.18j, k shows an example where the p symmetric pairs ($A1$,$A2$), ($B1$,$B2$), etc., are aligned horizontally after $\lceil \frac{p}{2} \rceil + 1$ executions of *Step 1y*. For such "pathological" examples, the computation of the y-coordinates will take $O(p \cdot n \log \log n)$ time. However, in most of the practical cases, no more than three iterations are necessary to fix the vertical symmetry and topological constraints, hence *Step 1y* runs typically in $O(n \log \log n)$ time.

1.3.1.2 The Computation of the Device Abscissae

Step 1x: The first traversal, called *initialization*, computes the block abscissae [31] leaving aside for the time being the symmetry constraints. For consistency sake, the pseudocode is given below:

```
insert bucket (0,0) in the priority queue;
      // this special bucket acts as a sentinel
for each index i in α (i = 1 to n)
  let l be the index in β of cell B_j = α_i ;
  find bucket pred_l whose index is the largest lesser
       than l ;
  x_j = length of pred_l ;
  insert bucket (l, x_j + w_j) into the queue;
  remove buckets with an index greater than l
         and a length lesser than or equal to x_j + w_j ;
end_for
remove all the buckets from the priority queue;
```

The worst-case complexity of *Step 1x* is $O(n \log \log n)$.

Example 2. Given the symmetry group $\{(A_1, A_2), B, (C_1, C_2), (D_1, D_2), (E_1, E_2), (F_1, F_2)\}$ consisting of five pairs of symmetric devices and one self-symmetric cell,

let ($F_1E_1BA_1A_2XE_2YD_1ZC_1C_2D_2F_2$, $F_1YD_1ZC_1C_2D_2E_1A_1A_2BXE_2F_2$) be a symmetric-feasible sequence-pair. The placement after the execution of *Step 1x* is shown in Fig. 1.19a. □

Step 2x: Next, the position of the symmetry axis (x_{symAxis}) is chosen. Different from [43], we employ a scheme for selecting the symmetry axis and initializing the abscissae of the devices before the next two traversals (steps) in the symmetry group such that the x-span of the group (and, ultimately, its area) is kept minimal (relative to the topological constraints) by the end of the evaluation algorithm. This axis selection scheme is described below.

The general idea of this step is to align the individual axes of the innermost symmetric pairs and to position the other pairs to leave enough space (but not more space than necessary) to satisfy the other horizontal topological constraints. First, a directed acyclic graph (DAG) is built from the S-F sequence-pair, showing the embedding relation between pairs along the Ox axis: each symmetric pair or self-symmetric device is a node in this DAG, the node of an inner pair being the successor of an outer one. E.g., for the example in Fig. 1.16b, a tree having the root (B,G) with three children (C, D), (A, A), and (F, F) is obtained, as shown in Fig. 1.20a. The embedding DAG for the placement in Fig. 1.18i is shown in Fig. 1.20b.

Initially, the DAG contains a node for each symmetric pair or self-symmetric cell. Since, in this moment, the vertical position of each cell is already known, the end points of the y-span intervals $[y_{cell}, y_{cell} + h_{cell}]$ of the cells in the symmetry group are iteratively inserted as keys in an initially empty priority queue. The order of insertion is given by sequence α, such that, at each moment, the priority queue contains the end points of the segments part of the right contour of this partial placement.

The construction of the embedding DAG will be illustrated using *Example 2*. First, the end points of the interval $[y_{F_1}, y_{F_1} + h_{F_1}]$=[0, 10] are inserted (see Fig. 1.19b1), followed by the end points of $[y_{E_1}, y_{E_1} + h_{E_1}]$=[6, 12]. Since this latter interval covers from the right the top margin of the former, key 10 is removed from the queue (see Fig. 1.19b2) and an arc from (F_1, F_2) to (E_1, E_2) is added to the DAG. Note that the segments visible from the right are [0, 6] from the F_1's interval and [6, 12] from the E_1's interval. The construction proceeds in this way, removing from the queue the keys covered by a new interval and adding arcs in the DAG each time segments from the right contour (belonging to cells in the symmetry group) are covered by the y-span interval of the new cell (see Fig. 1.19b3–b6). The DAG obtained for this example is shown in Fig. 1.19b7.

Afterward, the position of the symmetry axis is selected such that $x_{\text{symAxis}} = \max\left\{\frac{x_j+(x_k+w_k)}{2}\right\}$, for all nodes without successors (B_j, B_k) in the DAG. In our example, x_{symAxis}=14, corresponding to the pair (C_1, C_2). Then, each node without successors (B_j, B_k) will receive a value $d = x_{\text{symAxis}} - \frac{x_j+(x_k+w_k)}{2} \geq 0$ and the abscissae of B_j and B_k computed at *Step 1x* will be updated: $x_j = x_j + d$, $x_k = x_k + d$, therefore shifted to the right. Proceeding bottom-up in the embedding

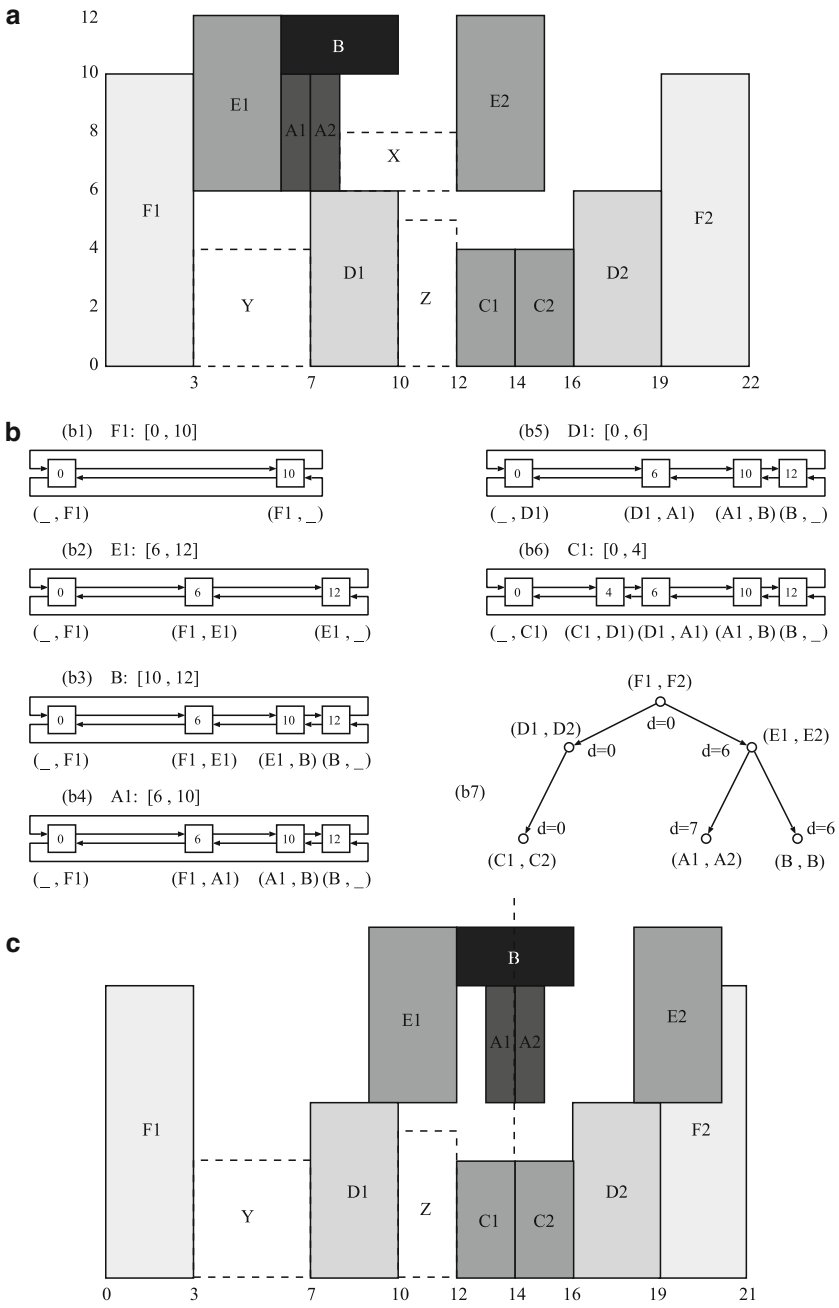


Fig. 1.19 (**a**) Placement after the initialization traversal (*Step 1x*) for the symmetric-feasible sequence-pair ($F_1E_1BA_1A_2XE_2YD_1ZC_1C_2D_2F_2$, $F_1YD_1ZC_1C_2D_2E_1A_1A_2BXE_2F_2$). The symmetry group is $\{(A_1, A_2), B, (C_1, C_2), (D_1, D_2), (E_1, E_2), (F_1, F_2)\}$. (**b**) Construction of the DAG showing the embedding relations in the symmetry group. (**c**) Update of the abscissae of the devices in the symmetry group at the end of *Step 2x*. The possible overlaps will eventually be fixed

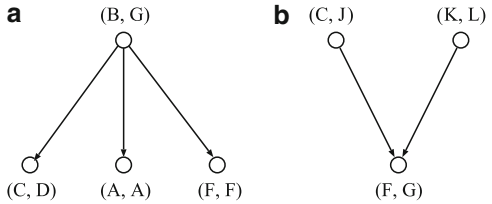


Fig. 1.20 (**a**) The embedding DAG for the placement in Fig. 1.16a. (**b**) The embedding DAG for the placement in Fig. 1.18i (*Example 1*)

DAG, each node will receive a d value equal to the minimum one among node's children (see Fig. 1.19b7); the abscissae of the node's cell(s) are similarly updated, as shown in Fig. 1.19c.

This operation is dominated by the building of the embedding DAG. The priority queue contains keys in the set $\{0, \ldots, H\}$, where H is the height of the placement. Since the key insertions and deletions take $O(\log \log H)$ each[12] [32], we may be tempted to consider $O(\log \log H)$ the worst-case time bound per iteration. However, this is not always true: when the bucket list of the priority queue grows, it can gain at most two new keys per iteration; but when it decreases in size, as many as $O(n)$ keys can be deleted.[13] However, the amortized time bound [51] is $O(\log \log H)$ per iteration. Intuitively, the reason is that each key can be deleted at most once for each time it is inserted. In an amortized analysis, the time required to perform a sequence of operations is averaged over all the operations performed. Since there are $p + s$ iterations (where p is the number of pairs and s is the number of self-symmetric cells), the overall time complexity of this step is $O((p + s) \log \log H)$ or, with a less tighter bound, $O(n \log \log H)$.

If, in addition, $H = \Theta(n)$, the overall complexity is $O(n \log \log n)$. Otherwise, the y-coordinates of the devices can be "normalized" by replacing each of them by its index in their increasingly-ordered set $\mathcal{S} = \bigcup_i \{y_i,\ y_i + h_i\}$. In this way, the y-coordinates can be considered, without loss of generality, integers in the range $[0, n]$ (n being the number of devices). Another important consequence of the "normalization" is the fact that the size of the priority queue will be kept minimal. For instance, in our illustrative example (Fig. 1.19a), after the sorting and elimination of duplicates, the set $\mathcal{S} = \bigcup_i \{y_i,\ y_i + h_i\} = \{0, 4, 6, 10, 12\}$ has five elements (for the cells in the symmetry group). Instead of inserting, say, for the cell E_1 the keys 6 and 12 (the end points of its y-span interval), we insert instead the keys 2 and 4, that is, their indexes in the set $\mathcal{S}$. Note that the height of the host binary tree of the priority queue (see Sect. 1.2.4) is reduced from 4 to 3 when normalized

[12] Actually, according to [32], the asymptotic upper bound is even tighter: $O(\log \log(\max_i h_i))$, where h_i are the heights of the devices. But in the worst case, $h_i = \Theta(H)$.

[13] Such a situation could occur if the priority queue contained $O(n)$ keys and in the next iteration the new cell had the height of the whole analog block; the priority queue would then be reduced to only two buckets having the keys 0 and H.

coordinates are used instead. The normalization is not done with a general purpose sorting algorithm (of worst-case complexity $O(n \log n)$) since then the complexity would be dominated by the sorting. Instead, the radix sort of complexity $O(n + H)$ [51] is used.[14] When using the radix sort for normalization, the complexity becomes $O(H + n \log \log n)$.

Step 3x: This traversal (*sweep-to-the-right*) is similar to the *initialization* step, but before inserting the bucket $(l, x_j + w_j)$ into the queue, if (B_k, B_j) is a symmetric pair and $d = 2x_{\text{symAxis}} - x_j - (x_k + w_k) > 0$, then x_j is increased with d. For the pair (D_1, D_2), for instance, $d = 2 \times 14 - 16 - (7 + 3) = 2$, therefore D_2's abscissa will be further increased with 2, becoming 18 (see Fig. 1.21a); note that (D_1, D_2) satisfy now the symmetry constraint.

Step 4x: At the end of the *sweep-to-the-right* traversal, some of the horizontal symmetry constraints may still be unsatisfied since some rightmost cells in the pairs were pushed further to the right (see, for instance, the pair (E_1, E_2) in Fig. 1.21a). Then, a third traversal of α (*sweep-to-the-left*) will fix all the symmetry constraints:

```
insert bucket (n+1,W) in the priority queue;
        // this bucket acts as a sentinel
for each index i in α (i = n to 1)
  let l be the index in β of cell B_j = α_i;
  find bucket succ_l whose index is the smallest greater
than l;
        // succ_l will always be found due to the sentinel bucket (n+1,W)
  x_j = (length of succ_l) - w_j;
  if (B_j, B_k) is a symmetric pair and d = x_j + (x_k + w_k) -
2x_symAxis > 0
  then x_j = x_j - d;
  insert bucket (l, x_j) into the queue;
  remove buckets with an index lesser than l
         and a length greater than or equal to x_j;
end_for
remove all buckets from priority queue;
```

With a similar reasoning, both *Step 3x* and *Step 4x* take $O(n \log \log n)$ time.

Note that in the absence of symmetry constraints, the evaluation algorithm reduces to *Step 1y* and *Step 1x*, therefore identical to [31].

Besides the LCS-based approach, which improves the overall complexity of the evaluation algorithm in comparison to [43], *Step 2x* ensures that the x-span of the symmetry group is kept minimal relative to the horizontal symmetry and topological constraints induced by the sequence-pair representation. First, the symmetric pairs are positioned about the symmetry axis at the minimal distance allowed by

[14] Sorting by key insertion in Johnson's priority queue – of complexity $O(n \log \log(H/n))$ [32] – can be used, as well.

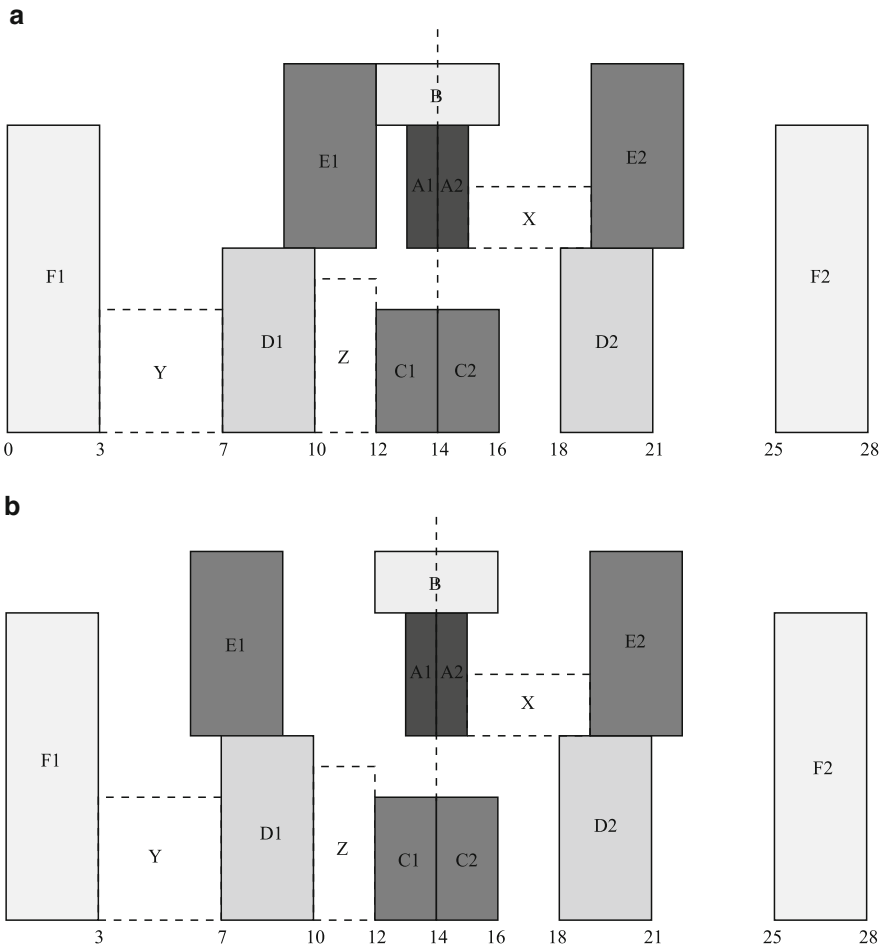


Fig. 1.21 Placements of the symmetric-feasible sequence-pair ($F_1E_1BA_1A_2XE_2YD_1ZC_1C_2D_2F_2$, $F_1YD_1ZC_1C_2D_2E_1A_1A_2BXE_2F_2$) after the *sweep-to-the-right* (*Step 3x*) and after the *sweep-to-the-left* traversal (*Step 4x*)

the topological constraints (as computed at *Step 1x*). Second, with the d-values of the embedding DAG's nodes, the leftmost cells of the pairs are positioned as close as possible from the symmetry axis. (From Fig. 1.19c, one can observe that no leftmost cell in the symmetric pairs can be brought closer to the symmetry axis without producing topological violations.) Afterward, the *sweep-to-the-right* step will shift the rightmost cells in the pairs just enough to satisfy the topological constraints (like E_2 in Fig. 1.21a), or symmetry constraints (like D_2), or both (like F_2). The subsequent *sweep-to-the-left* step will fix the remaining symmetry violations, like E_1 in Fig. 1.21b. Since the sequence-pair satisfies the property (1.1), there are no situations when two symmetric pairs are intertwined preventing each other to

align vertically about the same symmetry axis, so the algorithm ends after *Step 4x*. Moreover, since the abscissae are updated from the innermost pairs to the outermost ones, the symmetry group is packed around the symmetry axis as much as the topological constraints allow.

1.3.2 Handling Multiple Symmetry Groups

In order to handle an arbitrary number of symmetry groups, one must take the precaution that any two symmetry groups do not prevent each other from being correctly built. For instance, if two cells x, y belonging to different symmetry groups satisfy simultaneously the inequalities $\alpha^{-1}(x) < \alpha^{-1}(y) < \alpha^{-1}(\text{sym}(y)) < \alpha^{-1}(\text{sym}(x))$ and $\beta^{-1}(y) < \beta^{-1}(x) < \beta^{-1}(\text{sym}(x)) < \beta^{-1}(\text{sym}(y))$, then cell x results above cell y, whereas cell $\text{sym}(y)$ results above $\text{sym}(x)$, the two pairs preventing each other to align horizontally within the groups. Fortunately, it is possible to design the move set to avoid such situations. The easiest way is to prevent the cells from different groups to intermingle with each other in any of the sequences α and β, solution adopted in [43].

However, if the design requires embedded symmetry groups [59] – this being revealed by the schematic of the circuit, the move set can be modified to allow cells from one group between cells of another group in both sequences α and β simultaneously. Figure 1.22 shows a placement with three symmetry groups: *Group* $1 = \{(A_1, A_2),\ (B_1, B_2),\ (C_1, C_2),\ D,\ E\}$, *Group* $2 = \{(U_1, U_2),\ V\}$, and *Group* $3 = \{(X_1, X_2),\ (Y_1, Y_2),\ W,\ Z\}$, the second and

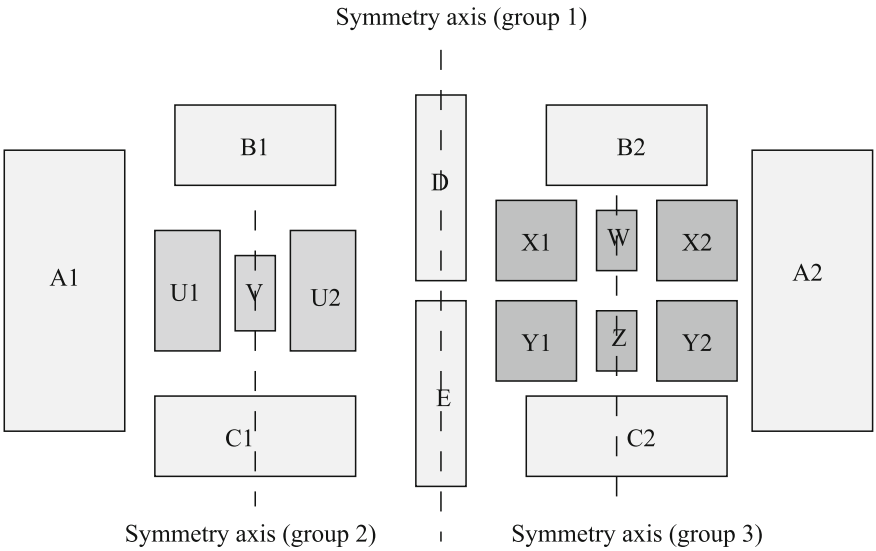


Fig. 1.22 Placement with three symmetry groups, the groups 2 and 3 embedded in group 1

the third groups being embedded in the first one. An encoding sequence-pair is: $(A_1 B_1 \mathbf{U_1 V U_2} C_1 DE B_2 \mathbf{X_1 Y_1 W Z X_2 Y_2} C_2 A_2,\ A_1 C_1 \mathbf{U_1 V U_2} B_1 EDC_2 \mathbf{Y_1 X_1 Z W Y_2 X_2} B_2 A_2)$. Note that the cells of the groups 2 and 3 (written with bold characters) are surrounded by the cells of *Group 1* in both sequences α and β (although it is not essential to occupy contiguous positions in any of the sequences). This constraint on the move set is sufficient to guarantee the feasibility of the placement and the completion of the placement algorithm.

With these remarks, the technique to handle an arbitrary number of symmetry groups is similar to the single-group case, but more traversals of the (symmetric-feasible) sequence-pair are necessary. The symmetry groups are processed one by one, after each iteration the symmetry constraints relative to one symmetry group being fixed, and the relative positions of the devices in that group being "frozen." If G is the number of symmetry groups, the complexity of the evaluation algorithm is basically $O(G \cdot n \log \log n)$. In the case of embedded symmetry groups, the inner groups must be processed before the outer ones, therefore the order of the group processing does matter. (An embedding DAG of the symmetry groups is used to order the group processing.) But if the cells of the groups do not intermingle in the sequence-pair, then the processing order of the groups is irrelevant. Note also that if the sequence-pair is not symmetric-feasible relative to all the groups, the algorithm may loop forever attempting to fix the symmetry constraints.

1.3.3 The Design of the Move Set

The move set of the simulated annealing algorithm was adapted to restrict the exploration of the sequence-pairs to the subset of those symmetric-feasible. To do this, it is sufficient to start the exploration with an initial sequence-pair, which is symmetric-feasible relative to all the symmetry groups [43]. Assuming for simplicity only one symmetry group, such a sequence-pair is, for instance:

$$(\alpha, \beta) = \left(a_1 \cdots a_p\, c_1 \cdots c_s\, b_p \cdots b_1 \cdots,\ a_1 \cdots a_p\, c_s \cdots c_1\, b_p \cdots b_1 \cdots\right)$$

where $(a_i, b_i),\ i = 1, \ldots, p$ are the pairs of symmetric cells and $c_j,\ j = 1, \ldots, s$ are self-symmetric cells. This sequence-pair corresponds to a placement where the pairs of symmetric cells are disposed in line, like embedded brackets, surrounding the self-symmetric cells which are disposed one on the top of the other. More general, we may pick randomly the order in α of all the devices in the symmetry group, and arrange in β their symmetric pairs in exactly the reverse order.

Afterward, the move set can be customized such that the property of symmetric-feasibility is preserved after each move. For instance, if two cells from distinct symmetric pairs are interchanged in the sequence α, then their symmetric counterparts must be interchanged as well in the sequence β; if a cell is moved, changing its position in the sequence α, its symmetric pair must be moved too in the sequence β, and the range of possible positions of this latter move depends on the move of

the former cell. Device rotations and mirroring are also affecting simultaneously two symmetric cells. Note that the moves of the cells in the asymmetric component of the circuit (interchanges and changes of position in both sequences, device rotations) are unrestricted (except for the move amplitude due to the cooling temperature of the annealing), so the tool works also in the absence of symmetry constraints. More complex moves operating with entire symmetry groups are also performed – although with a low probability, decreasing with the temperature.

1.4 Topological Placement with Symmetry Constraints Using Other Layout Representations

1.4.1 A Comparative Overlook on Transitive Closure Graphs

The transitive closure graph (TCG) representation, introduced by Lin and Chang [39], is based on two directed graphs – the horizontal and vertical transitive closure graphs, denoted C_h and C_v – both having a node for each cell, the edges corresponding to the topological relations between cells. For instance, the arc (A, B) in C_h means that A is to the left of B; the same arc in C_v means that A is below B. Each pair of vertices must be connected by exactly one edge either in C_h or in C_v.

Since Lin and Chang addressed the placement problem with symmetry constraints within the TCG representation [46], this section comments on whether working with the TCG sequence (TCG-S) instead of S-F sequence-pairs could lead to a better exploration of the solution space in terms of speed and/or quality.

A key remark is that, actually, *the TCG and sequence-pair representations are equivalent*. As noticed by Zhang and El-Masry [60] (but also by the authors of TCG), the β sequence in the sequence-pair is the topological sorting of both graphs C_h and C_v. Indeed, A precedes B in sequence β if there is an arc $A \longrightarrow B$ either in C_h or in C_v. Normally, one can construct different topological sorting sequences based on the same graph; but both graphs C_h and C_v determine a unique topological sorting sequence which can be built in $O(n \log n)$ time employing a sorting algorithm, where the comparison rule is whether the source vertex is ahead of the sink vertex for any directed edge in C_h or C_v. Similarly, the sequence α is the topological sorting sequence of the graphs C_h and C_v, the latter with the arcs reversed [60].

From the arguments above, it follows that given a TCG for n cells, one can build the corresponding sequence-pair in $O(n \log n)$ time. Reciprocally, it takes $O(n^2)$ time to transform a given sequence-pair into a TCG. Note that the latter complexity cannot be improved since the total number of arcs in C_h and C_v is $\Theta(n^2)$.

In conclusion, the TCG and sequence-pair representations are equivalent, so the sizes of their solution spaces are the same, that is $(n!)^2$. Obviously, one can introduce a symmetric-feasibility condition within the TCG representation: e.g., TCG is symmetric-feasible if its corresponding sequence-pair is symmetric-feasible in the

sense of condition (1.1), or defining direct conditions in the two graphs C_h and C_v.[15] However, we cannot find any advantage relative to the size of the exploration space, or to the solution quality – both representations being P-admissible [29]. On the other hand, the proposed evaluation algorithms for TCG-S (executed in each inner-loop iteration of the simulated annealing, evaluating the layout cost after each move) have quadratic complexity [60], whereas Sect. 1.3.1 presented an evaluation algorithm for S-F sequence-pairs of better complexity. Therefore, using S-F sequence-pairs is more advantageous in terms of computational speed.

1.4.2 A Comparative Overlook on Tree Representations of the Layout

According to [34], a placement configuration of n rectangular blocks can be represented by an ordered tree (O-tree), that is a tree with $n + 1$ nodes, encoded by $(\mathcal{T}, \pi)$, where $\mathcal{T}$ is a $2n$-bit string identifying the branching structure of the tree relative to a traversal order (a "0" corresponds to descending an edge, while an "1" – to subsequently ascending that edge), and π is a permutation of the block names. Figure 1.23a shows an O-tree having the encoding $(\mathcal{T}, \pi)$=(00110100011011, *adbcegf*).

Let $T = (T_1, \ldots, T_k)$ be an O-tree, where $T_1, \ldots, T_k$ are its subtrees relative to the root. The O-tree can be transformed recursively into a binary tree $B(T)$ as follows:

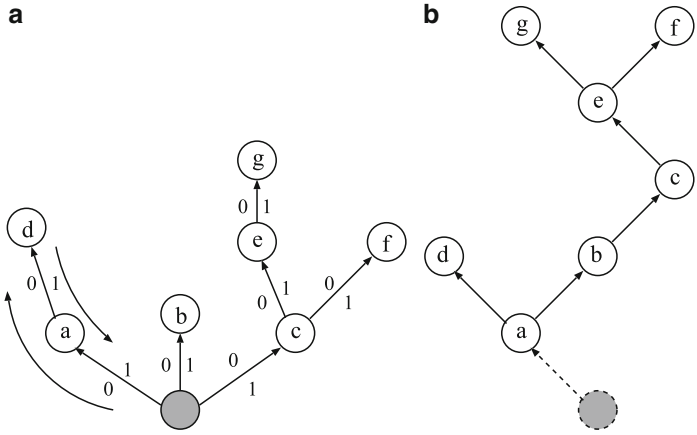


Fig. 1.23 (**a**) O-tree representation; (**b**) binary tree representation

[15] The feasibility of TCG in symmetry point of view is defined differently in [46], but the modification of the size of the exploration space is not addressed.

a. If $k = 0$, $B(T)$ is empty;
b. If $k > 0$, the root of $B(T)$ is the root of T_1; the left subtree of $B(T)$ is $B(T_1)$; the right subtree of $B(T)$ is $B(T')$, where $T' = (T_2, \ldots, T_k)$ is the initial O-tree less the subtree T_1.

The binary tree derived from the O-tree in Fig. 1.23a is shown in Fig. 1.23b, the root of the binary tree becoming obsolete. As the inverse transformation is straightforward, it follows that there is a one-to-one correspondence between the sets of O-trees and of binary trees having one node less.

The tree representations of the layout are important in our context since their number is lesser than the number of sequence-pair encodings, yielding thus a smaller exploration space for the block placement problems. Indeed, since in a binary tree representation the nodes are labeled with the names of the cells, the number of codes is $b_n \cdot n!$, where b_n is the number of unlabeled binary trees with n nodes – known as the nth *Catalan number* [37]: $b_n = \frac{1}{n+1}\binom{2n}{n}$, where $\binom{\alpha}{n} = \frac{\alpha(\alpha-1)\cdots(\alpha-n+1)}{n!}$.

The relation between the number of tree topological representations and the number of sequence-pairs is given by the following

Lemma 2. *The number of labeled binary trees with n nodes is smaller than the number of n-block sequence-pairs.*

Proof. Assuming $n \geq 3$, for any integer p such that $1 < p < n$, we have the inequality $\frac{n+p}{p} < n - p + 2$. Indeed, by trivial computations, the inequality above is equivalent to $0 < (n-p)(p-1)$. Substituting $p = 2, 3, \ldots, n-1$ in the inequality, we obtain $\frac{n+2}{2} < n$, $\frac{n+3}{3} < n-1$, $\ldots$, $\frac{2n-1}{n-1} < 3$. Multiplying, $\frac{n+1}{1} \cdot \frac{n+2}{2} \cdot \frac{n+3}{3} \cdots \frac{2n-1}{n-1} \cdot \frac{2n}{n} < (n+1) \cdot n \cdot (n-1) \cdots 3 \cdot 2$.

This is equivalent to $\binom{2n}{n} < (n+1)!$, or $b_n < n!$. Multiplying both sides of this inequality by $n!$, we obtain the stated result. When $n = 1, 2$, the numbers of trees and sequence-pairs are equal (to 1 and, respectively, 4). □

Although the number of tree representations is smaller than the sequence-pair encodings, the number of *symmetric-feasible* sequence-pairs may be smaller than the number of tree representations when the symmetry groups are dominant (in terms of number of cells). For instance, if all the devices of a circuit form a symmetry group ($n = 2p$), from Lemma 1 the number of symmetric-feasible sequence-pairs is upper-bounded by $(2p)!$, whereas the number of trees is $b_{2p} \cdot (2p)!$ – therefore, at least twice larger. On the contrary, when the asymmetric part of the circuit is dominant, the number of tree representations becomes smaller.

A placement approach for analog layout with symmetry constraints, based on the exploration of O-trees, was proposed in [44]. The main idea was to build horizontal and vertical constraint graphs from the current O-tree: if the horizontal constraint graph has cycles or if the vertical constraint graph has positive cycles, the O-tree is infeasible in symmetry point of view and hence disregarded. The complexity of these tests is quadratic.

However, a better strategy is to limit the exploration to a subset of binary tree representations yielding layouts automatically satisfying the given symmetry constraints, exactly as we did in the case of sequence-pair encodings. The next section will introduce such a subset of binary tree representations whose size is smaller than both the number of symmetric-feasible sequence-pairs and the number of tree representations.

1.4.2.1 Symmetric-Feasible Binary Trees

A binary tree layout representation, whose nodes represent the rectangular cells in a placement configuration, induces the following vertical (y-) and horizontal (x-) positioning constraints [36]:

a. Each cell whose node is in the left subtree is above the cell whose node is the parent;
b. If the y- projections of two cells are overlapping, the cell whose node is visited first in a preorder traversal of the tree (i.e., visit any node before its left and right subtrees) is to the left of the cell whose node is visited the second.

The nodes of a binary tree can be visited in preorder, inorder, or postorder. A preorder traversal of a binary tree starts from the root, visiting the current node and then recursively visiting its left subtree, followed by its right subtree. An inorder traversal visits the current node in between recursively visiting its left and right subtrees. It is well known that the pair of preorder and inorder traversals uniquely determine the binary tree. Several algorithms building the tree from its traversals in optimal time and space were proposed (e.g., [61, 62]).

Remark: The pair of sequences (inorder, preorder) traversals should not be mistakenly confused with the sequence-pair representation (α, β) proposed by Murata et al. [29]. There are sequences α and β that do not correspond to the inorder and preorder traversals of *any* binary tree: such a sequence-pair is, for instance, (CAB, ABC). More general, for any $n > 3$ one can build a pair of cell permutations $(\ \ldots\ CAB,\ \ldots\ ABC)$ that does not correspond to the (inorder, preorder) traversals of any binary tree. This can be seen as another proof of the fact that the number of sequence-pairs is larger than the number of binary trees for $n \geq 3$.

A binary tree representation is *symmetric-feasible* if its pair of (inorder, preorder) traversal has the following property [45]: for any distinct nodes (cells) A, B in a symmetry group,

$$A \overset{\text{inorder}}{\prec} B \iff \text{sym}(B) \overset{\text{preorder}}{\prec} \text{sym}(A) \tag{1.2}$$

i.e., the node A *precedes* node B in the inorder traversal of the binary tree if and only if the node sym(A) – corresponding to cell A's symmetric pair – *succeeds*

node sym(B) in the preorder traversal. In addition, any two nodes A, B belonging to different symmetry groups cannot satisfy the inequalities[16]

$$A \overset{\text{inorder}}{\prec} B \overset{\text{inorder}}{\prec} \text{sym}(B) \overset{\text{inorder}}{\prec} \text{sym}(A)$$
$$B \overset{\text{preorder}}{\prec} A \overset{\text{preorder}}{\prec} \text{sym}(A) \overset{\text{preorder}}{\prec} \text{sym}(B)$$

Similar as in Sect. 1.3.1, one can develop evaluation algorithms for this subset of *symmetric-feasible* (S-F) binary tree representations using any of the data structures presented in Sect. 1.2 (see, for instance, [45, 53]).

1.4.2.2 The Design of the Move Set

The design of the move set when operating with topological representations based on trees [34, 35] was a topic insufficiently addressed before. The problem is to conceive a set of moves such that any code in the topological representation is theoretically reachable by applying a finite sequence of moves from any given code. Besides cell interchanges that affect only the labels of the nodes, a typical move that modifies the tree structure is to detach a subtree and re-attach it to another available node [34]. This move cannot be used in our framework since the preservation of property (1.2) is difficult to accomplish. Two less obvious move sets for the general binary tree representation will be presented below. Due to the natural correspondence between forests of rooted trees and binary trees [37], the equivalent moves for O-trees can be easily derived.

(a) Move sets for binary trees with provable reachability

1. To transform a left-parenthesized product $(...((a_0a_1)a_2)...a_n)$ into a right-parenthesized product $(a_0(a_1(...(a_{n-1}a_n)...)))$, one may apply a sequence of basic transformations $((xy)z) \mapsto (x(yz))$, involving three subexpressions x, y, z [37]. A binary tree whose nodes have two children each can be represented by parenthesized products.[17] The transformation $((xy)z) \mapsto (x(yz))$ can then be applied to binary trees replacing the subexpressions with subtrees: $((S_1S_2)S_3) \mapsto (S_1(S_2S_3))$. This operation may be thought of as "sliding" a right-descendent subtree of a left-descendent node past its parent into a left-descendent subtree of the corresponding right descendent: $\curlyvee \mapsto \curlyvee$. Since the parenthesized products can be represented as the vertices of a polyhedron – a Stasheff polytope [63] – where the neighbor vertices differ from one another by the above transformation (or its inverse), a similar property is valid for the binary trees whose nodes have two children each.

[16] This second condition eliminates those trees where cell A results above cell B, whereas cell sym(B) results above sym(A), the two pairs preventing each other to be aligned horizontally within the groups.

[17] See the parenthesized notation of trees in [37].

A binary tree representation is slightly different in the sense that each node has *at most* two children (rather than *exactly* two children). However, if the *NULL* pointers (or references) are viewed as dummy children, the nodes in the binary tree representation will have two children each, and the basic transformation described above can be applied, along with its inverse. Moreover, these transformations ensure the reachability of any unlabeled binary tree since they represent vertices in the Stasheff polytope [63] – which are edge-connected.

Interchanging the labels of two nodes does not modify the branching structure of the binary tree representation. This swap move ensures the reachability of any labeled binary tree having a given branching structure, since a permutation of node labels can be decomposed into a sequence of interchanges. Therefore, the structural transformation $((S_1S_2)S_3) \mapsto (S_1(S_2S_3))$ and the interchange of node labels ensure the complete exploration of the binary tree codes.

2. It is well known [64] that there is a one-to-one mapping between the set of rectangular-grid paths connecting two opposite corners of a square and which do not cross the diagonal, and the set of unlabeled binary trees with n nodes, where n is the number of grid units of the square side (see Fig. 1.24). If the horizontal unit segments of such a path are denoted by E (from *East*) and the vertical ones by N (from *North*), a path as described above can be represented by a sequence of n Es and n Ns. In order to prevent the diagonal crossing, in any leading subsequence the number of N's must not be larger than the number of Es. Therefore, the branching structure of the tree can be uniquely described by such a sequence of Es and Ns.[18]

   A move that modifies the branching structure of the tree is the interchange of an E segment and an N segment in the corresponding path – called a

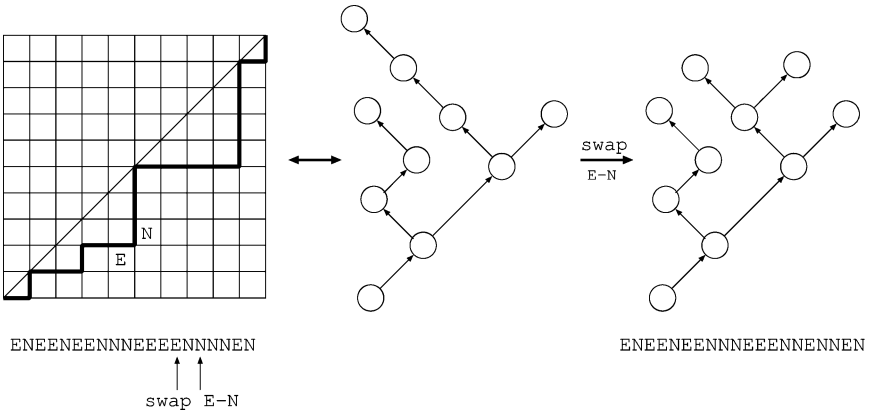


Fig. 1.24 Correspondence between grid paths in a square and unlabeled binary trees

[18] The O-tree encoding in [34] is similar, but this path analogy was not mentioned.

swap E-N – such that the diagonal crossing is prevented (Fig. 1.24). The $2n$-long path γ_0: E$\cdots$EN$\cdots$N can be easily transformed into another path γ by a finite sequence of E-N interchanges: for instance, if the first N occupies position $k_1 (\leq n)$ in γ, the k_1th and the $(n+1)$th segments in γ_0 are swapped; if the second N occupies position $k_2 (k_1 < k_2 \leq n)$ the k_2th and the $(n+2)$th segments are swapped, and so on. The transformation $\gamma_1 \mapsto \gamma_2$ between two arbitrary paths can be obtained as a combination of the transformations $\gamma_1 \mapsto \gamma_0$ (which is the inverse of $\gamma_0 \mapsto \gamma_1$) and $\gamma_0 \mapsto \gamma_2$. Consequently, the cell interchange together with the swap E-N ensure the reachability of any labeled binary tree and, therefore, the complete exploration of the binary tree codes.

Our placement tool employs the second move set described above. In addition, the move set is adapted to preserve the symmetric-feasibility property (1.2) – as will be explained below.

(b) The move set in the presence of symmetry constraints

At the beginning of the simulated annealing, the placement tool builds a binary tree satisfying the property (1.2) for each symmetry group [45]. Afterward, each move is performed such that the property (1.2) be preserved. In the current implementation, the moves are of three types. The first two moves are cell interchanges and swaps E-N. The presence of symmetry constraints impose some restrictions (discussed below) on the use of these moves to preserve the feasibility of the codes in symmetry point of view. The third type of move is specific only to placement configurations containing several symmetry groups.

1. *Interchange of two cells*

When the cells belong to the same symmetry group but they are not the symmetric pairs of each other, the cell interchange must be accompanied by the swap of their symmetric pairs to preserve the property (1.2). For the same reason, interchanges between cells belonging to different symmetry groups, or when only one cell belongs to a symmetry group are allowed only between cells whose nodes are in a parent–child relation. This move – having a constant complexity – ensures the reachability of any S-F binary tree having a given branching structure, since a permutation of node labels can be decomposed in a sequence of such interchanges.

2. *Move of cells*

When a cell from the asymmetric part of the circuit is selected, the swap E-N is performed exactly like in the absence of symmetry. When the randomly selected cell is part of a symmetry group, two simultaneous swaps E-N of linear complexity are performed – one for the cell and the other for its symmetric pair – based of the sequences of traversals (inorder, preorder). The binary tree is locally modified around the nodes corresponding to the chosen cell and its symmetric pair such that the property (1.2) be maintained.

3. *Move of symmetry groups*

In placement problems with several symmetry groups, more complex moves (of quadratic complexity) are also performed – albeit with a low probability that decreases with the temperature. They modify the position and the structure of an entire symmetry group: the nodes corresponding to the devices belonging to a symmetry group are extracted from the current binary tree, and a new symmetric-feasible subtree is built with them. Afterward, this binary subtree is attached in the main tree.

In addition to these three types of moves, changes of cell orientation (rotations and mirror transformations) are also performed. The changes of orientation take into account the different forms of pair symmetry – *mirror* or *perfect*, the pairs of symmetric cells having mirrored or identical orientations, as well as the *self-symmetry* – when a device presenting geometric symmetry shares the same axis with the group [7].

1.5 Experimental Results

A prototype placement tool for analog layout using selectable exploration algorithms has been implemented in C++. The tool uses the simulated annealing algorithm as the combinatorial optimization engine. In order to ensure a comparative evaluation as correct as possible, the cost function, the simulated annealing cooling schedule, and the inner-loop criterion were set identical during testing for all the placement algorithms.

The tool can operate both with different topological representations (sequence-pairs and trees) and different code evaluation algorithms, using the data structures from Sect. 1.2. Besides symmetry constraints, the tool handles systematically-induced device mismatches, alignment constraints, and performs shape optimizations for parametric cells and for "soft" cells like capacitors, with the aspect ratio varying continuously between given limits. In addition, for the purpose of a complete comparative assessment, a complementary placement algorithm based on the traditional absolute representation has been embedded in the tool as well.

Figure 1.25 shows the placement for a telescopic opamp with gain-boost amplifiers. Figure 1.26 displays the placement for a frequency divider with selectable ratio having five groups of symmetry.

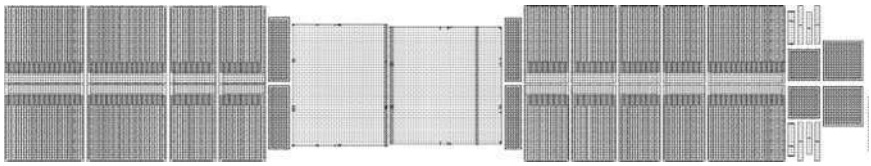


Fig. 1.25 Placement for a telescopic opamp with gain-boost amplifiers

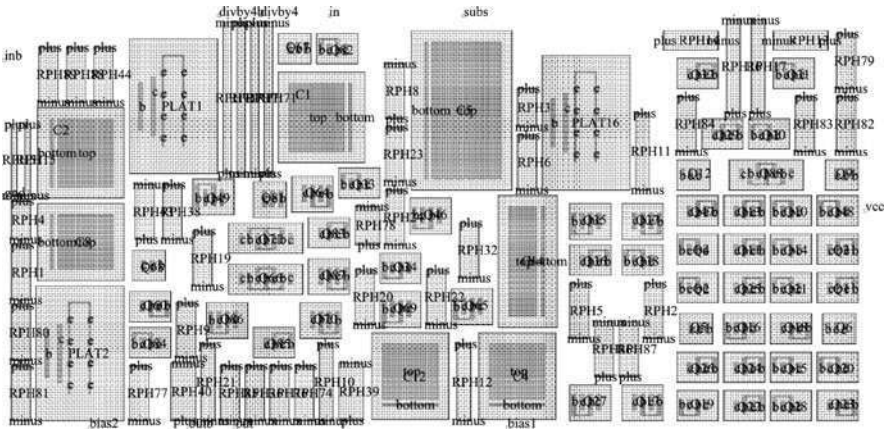


Fig. 1.26 Placement for a frequency divider with selectable ratio (2/4) having five symmetry groups

Table 1.1 Placement results

Design	Constraints	Nr. cells	Area [μm × μm]	Time [min]
Gain-boost amplifier	Sym.,dev. matching	17	71.2 × 68.0	0.2
Telescopic opamp with gain-boost amplifiers	Sym.,dev. matching, and four soft cells	36	527.2 × 96.0	1.6
Programmable capacitor block 1	Six soft cells	28	175.2 × 115.2	0.5
Programmable capacitor block 2	12 soft cells	34	220.8 × 186	1.1
15 MHz buffer	–	64	189.5 × 250.5	3.2
Amplifier with selectable gain	–	79	191.5 × 251.0	4.3
Bias current generator	–	85	237 × 186	5.0
Charge pump	Sym.,dev. matching	98	220.5 × 333.0	12.6
Limiter (17 ÷ 500 MHz)	Sym.,dev. matching, and 14 soft cells	111	177.5 × 375.0	16.9
Frequency divider with selectable ratio	Five sym. groups	116	350 × 147	21.0

Table 1.1 displays only a part of the experimental results carried out on a SUN Blade 100 workstation. The test benchmarks are analog blocks, several containing symmetry groups of devices, components of a spread spectrum transceiver used in wireless modems. Column 2 shows the type of constraints present in the design and column 3 displays the number of devices. Values of the placement area and CPU time are given only for the algorithm presented in Sect. 1.3.1. Evaluation algorithms employing the other data structures presented in Sect. 1.2 are somewhat slower, but not very significantly. The slowest appears to be the evaluation using segment trees (Sect. 1.2.1); however, the evaluations using red–black interval trees and deterministic skip lists (Sects. 1.2.2 and 1.2.3) are almost as fast as the one using priority queues.

The placement algorithm based on the exploration of symmetric-feasible binary trees is typically faster than the algorithm based on symmetric-feasible sequence-pairs (although the layout quality seems to be poorer). The difference in running times is not unexpected since the solution space of the binary tree representation is always smaller than the solution space of the sequence-pair representation for any placement problem, with or without symmetry constraints. The quality of the layout though depends also on the move set, on how uniformly the solution space is explored. The evaluation algorithms based on symmetric-feasible sequence-pairs seem to be better.

The running times are typically higher than those obtained by other topological placement tools operating on examples without symmetry (e.g., [31]). This is not unexpected since, when the cells belong to symmetry groups, their moves within the simulated annealing optimizer are, typically, a few times more computationally expensive; in addition, there are more traversals of the topological representation than in the absence of symmetry constraints. Restricting the moves within the subset of symmetric-feasible codes is costly for sure, but this strategy is significantly better than the exploration of the entire solution space of the topological representation employed.

The experiments also led to another conclusion: *all* the techniques exploring symmetric-feasible topological representations exhibit a significantly better performance, at least in terms of computational effort (but sometimes also in terms of placement quality), than using the more traditional absolute representation. Also the tuning of the simulated annealing optimizer was easier when using topological representations.

1.6 Conclusions

This chapter has given an overview of topological placement techniques handling symmetry constraints for analog layout synthesis. Different from most of the existent tools based on a simulated annealing optimization operating on absolute representations of the layout, this chapter has explored the use of some topological representations not restricted to slicing structures, where symmetry constraints – typical in analog placement – are directly taken into account during the exploration of the solution space.

References

1. R.A. Rutenbar and J. Cohn, Layout tools for analog ICs and mixed-signal SoCs: A survey, *Proc. Int. Symp. on Physical Design*, pp. 76–83, San Diego CA, April 2000
2. G.G.E. Gielen and R.A. Rutenbar, Computer-aided design of analog and mixed-signal integrated circuits, *Proceedings of the IEEE*, 88(12):1825–1852, 2000

3. D. Leenaerts, G.G.E. Gielen, and R.A. Rutenbar, CAD solutions and outstanding challenges for mixed-signal and RF IC design, in *Proc. IEEE/ACM Int. Conf. on Comp. Aided Design*, pp. 270–277, San Jose CA, Nov. 2001
4. J. Kuhn, Analog module generators for silicon compilation, in *VLSI System Design*, 1987
5. G. Benkeer, J. Conway, G. Schrooten, and A. Slenter, Analog CAD for consumer ICs, in *Analog Circuit Design*, J. Huijsing, R. van der Plassche, and W. Sansen (eds.), Norwell, MA: Kluwer, 1993, pp. 347–367
6. B.S. Baker, E.G. Coffman, and R.L. Rivest, Orthogonal packings in two dimensions, *SIAM J. Comput.*, 9(4):846–855, 1990
7. J. Cohn, D. Garrod, R. Rutenbar, and L. Carley, *Analog Device-Level Automation*, Norwell, MA: Kluwer, 1994
8. M. Kayal, S. Piguet, M. Declerq, and B. Hochet, SALIM: a layout generation tool for analog ICs, in *Proc. IEEE Custom Integrated Circuits Conf.*, pp. 7.5.1–4, 1988
9. S.W. Mehranfar, STAT: a schematic to artwork translator for custom analog cells, in *Proc. 1990 IEEE Custom Integrated Circuits Conf.*, pp. 30.2.1–3, 1990
10. S.W. Mehranfar, A technology-independent approach to custom analog cell generation, *IEEE J. Solid-State Circuits*, SC-26(3):386–393, 1991
11. E. Malavasi, J.L. Ganley, and E. Charbon, Quick placement with geometric constraints, in *Proc. IEEE Custom Integrated Circuits Conf.*, pp. 561–564, 1997
12. J. Cohn, D. Garrod, R. Rutenbar, and L. Carley, KOAN/ANAGRAM II: new tools for device-level analog layout, *IEEE J. of Solid-State Circuits*, SC-26(3):330–342, 1991
13. E. Malavasi, E. Charbon, E. Felt, and A. Sangiovanni-Vincentelli, Automation of IC layout with analog constraints, *IEEE Trans. CAD of IC's and Syst.*, 15(8):923–942, 1996
14. K. Lampaert, G. Gielen, and W. Sansen, A performance-driven placement tool for analog integrated circuits, *IEEE J. Solid-State Circ.*, 30(7):773–780, 1995
15. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi, Optimization by simulated annealing, *Science*, 220(4598):671–680, 1983
16. J. Cohoon and W. Paris, Genetic placement, *IEEE Trans. on Comp.-Aided Design of IC's and Systems*, 6(6):956–964, 1987
17. W.-J. Sun and C. Sechen, Efficient and effective placement for very large circuits, *IEEE Trans. on Comp.-Aided Design of IC's and Systems*, 14(3):349–359, 1995
18. J. Rijmenants, J.B. Litsios, T.R. Schwarz, and M. Degrauwe, ILAC: an automated layout tool for analog CMOS circuits, *IEEE J. of Solid-State Circuits*, SC-24(2):417–425, 1989
19. L. Zhang, R. Raut, Y. Jiang, and U. Kleine, Two-stage placement for VLSI analog layout designs, *IEE Proc. Circuits, Devices and Syst.*, 153(3):274–280, 2006
20. L. Zhang, R. Raut, Y. Jiang, and U. Kleine, Placement algorithm in analog layout designs, *IEEE Trans. CAD of IC's and Syst.*, 25(10):1889–1903, 2006
21. S. Kouda, C. Kodama, and K. Fujiyoshi, Improved method of cell placement with symmetry constraints for analog IC layout design, in *Proc. Int. Symp. on Physical Design*, pp. 192–199, San Jose CA, April 2006
22. N. Lourengo, M. Vianello, J. Guilherme, and N. Horta, LAYGEN – Automatic layout generation of analog ICs from hierarchical template descriptions, in *Research in Microelectronics and Electronics*, 2006
23. N. Jangkrajarng, L. Zhang, S. Bhattacharya, N. Kohagen, and R. Shi, Template-based parasitic-aware optimization and retargeting of analog and RF integrated circuit layouts, in *Proc. IEEE Int. Conf. on Comp.-Aided Design*, pp. 342–348, San Jose CA, Nov. 2006
24. A. Grebne, *Bipolar and MOS Integrated Circuit Design*, NY: Wiley, 1984
25. D.W. Jepsen and C.D. Gellat Jr., Macro placement by Monte Carlo annealing, in *Proc. IEEE Int. Conf. on Comp. Design*, pp. 495–498, Nov. 1983
26. R. Otten, Complexity and diversity in IC layout design, in *Proc. IEEE Int. Symp. Circuits and Computers*, 1980
27. D.F. Wong and C.L. Liu, A new algorithm for floorplan design, in *Proc. 23rd ACM/IEEE Design Automation Conf.*, pp. 101–107, 1986
28. E. Malavasi, E. Charbon, G. Jusuf, R. Totaro, and A. Sangiovanni-Vincentelli, Virtual symmetry axes for the layout of analog IC's, in *Proc. 2nd ICVC*, pp. 195–198, Seoul, Korea, Oct. 1991

29. H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani, VLSI module placement based on rectangle-packing by the sequence-pair, *IEEE Trans. CAD of IC's and Syst.*, 15(12): 1518–1524, 1996
30. X. Tang, R. Tian, and D.F. Wong, Fast evaluation of sequence pair in block placement by longest common subsequence computation, in *Proc. Design Aut. & Test in Europe*, pp. 106–111, Paris, March 2000
31. X. Tang and D.F. Wong, FAST-SP: A fast algorithm for block placement based on sequence pair, in *Proc. Asia-S. Pacific Design Aut. Conf.*, pp. 521–526, Yokohama, Japan, 2001
32. D.B. Johnson, A priority queue in which initialization and queue operations take $O(\log \log D)$ time, *Mathematical Systems Theory*, 15:295–309, 1982
33. S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani, Module packing based on the BSG-structure and IC layout applications, *IEEE Trans. on Comp.Aided Design of IC's and Systems*, 17(6):519–530, 1998
34. P.-N. Guo, C.-K. Cheng, and T. Yoshimura, An O-tree representation of non-slicing floorplan and its applications, in *Proc. 36th Design Aut. Conf.*, pp. 268–273, New Orleans LA, June 1999
35. Y.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu, B*-Trees: a new representation for non-slicing floorplans, in *Proc. 37th Design Aut. Conf.*, pp. 458–463, Los Angeles CA, June 2000
36. F. Balasa, Modeling non-slicing floorplans with binary trees, in *Proc. IEEE Int. Conf. on Comp.-Aided Design*, pp. 13–16, San Jose CA, Nov. 2000
37. D.E. Knuth, *The Art of Computer Programming*, vol. 1 (3rd ed.), MA: Addison-Wesley, 1997
38. X. Hong, G. Huang, Y. Cai, J. Gu, S. Dong, C.-K. Cheng, and J. Gu, Corner block list: an effective and efficient topological representation of non-slicing floorplan, in *Proc. IEEE Int. Conf. on Comp.-Aided Design*, pp. 8–12, San Jose CA, Nov. 2000
39. J.-M. Lin and Y.-W. Chang, TCG: A transitive closure graph-based representation for non-slicing floorplans, in *Proc. 38th Design Aut. Conf.*, June 2001
40. M. Sarrafzadeh and C.K. Wong, *An Introduction VLSI Physical Design*, NY: McGraw Hill, 1996
41. N. Sherwani, *Algorithms for VLSI Physical Design Automation*, 3rd ed., Boston: Kluwer, 1999
42. S.M. Sait and H. Youssef, *VLSI Physical Design Automation: Theory and Practice*, Singapore: World Scientific, 1999
43. F. Balasa and K. Lampaert, Symmetry within the sequence-pair representation in the context of placement for analog design, *IEEE Trans. CAD of IC's and Syst.*, 19(7):721–731, 2000
44. Y.-X. Pang, F. Balasa, K. Lampaert, and C.-K. Cheng, Block placement with symmetry constraints based on the O-tree non-slicing representation, in *Proc. 37th ACM/IEEE Design Aut. Conf.*, pp. 464–467, Los Angeles CA, June 2000
45. F. Balasa, S.C. Maruvada, and K. Krishnamoorthy, On the exploration of the solution space in analog placement with symmetry constraints, *IEEE Trans. CAD of IC's and Syst.*, 23(2): 177–191, 2004
46. J.-M. Lin and Y.-W. Chang, TCG-S: Orthogonal coupling of P-admissible representations for non-slicing floorplans, in *Proc. 39th Design Aut. Conf.*, June 2002
47. S. Dong, J. Liu, and X. Hong, Signal-path driven symmetry constraint for analog layout in SOI technology, manuscript, 2010
48. J.L. Bentley, Algorithms for Klee's rectangle problem, Res. Report, Pittsburgh, PA: Carnegie-Mellon University, 1977
49. F.P. Preparata and M.I. Shamos, *Computational Geometry*, Berlin: Springer, 1985
50. F. Balasa, S.C. Maruvada, and K. Krishnamoorthy, Efficient solution space exploration based on segment trees in analog placement with symmetry constraints, in *Proc. IEEE Int. Conf. Comp.-Aided Design*, pp. 497–502, San Jose CA, Nov. 2002
51. T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, NY: McGraw-Hill, 1990
52. L.J. Guibas and R. Sedgewick, A diochromatic framework for balanced trees, in *Proc. 19th Annual Symposium on Foundations of Computer Science*, pp. 8–21, 1978
53. S.C. Maruvada, K. Krishnamoorthy, F. Balasa, L.M. Ionescu, Red-black interval trees in device-level analog placement, *IEICE Trans. on Fundamentals of Electronics, Communications and Computer Sciences*, E86-A(12):3127–3135, 2003 (Special section on VLSI Design and CAD Algorithms), Japan

54. D.E. Knuth, *The Art of Computer Programming*, vol. 3, MA: Addison-Wesley, 1973
55. W. Pugh, Skip lists: A probabilistic alternative to balanced trees, *Comm. of the ACM*, 33(6):668–676, 1990
56. J. Culberson, J.I. Munro, Explaining the behavior of binary search trees under prolonged updates: a model and simulation, *The Computer Journal*, 32(1):68–75, 1989
57. J.I. Munro, T. Papadakis, R. Sedgewick, Deterministic skip lists, *Proc. 3rd Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 367–375, Orlando FL, Jan. 1992
58. T. Papadakis, *Skip Lists and Probabilistic Analysis of Algorithms*, Ph.D. Dissertation, University of Waterloo, 1993
59. K. Lampaert, *Analog Layout Generation for Performance and Manufacturability*, Ph.D. Thesis, K.U. Leuven, Belgium, Jan. 1998
60. L. Zhang and Ezz. I. El-Masry, Graph-based placement algorithm for analog LSI/VLSI physical designs, manuscript, 2006
61. A. Andersson and S. Carlsson, Construction of a tree from its traversals in optimal time and space, *Inform. Processing Letters*, 34:21–25, 1990
62. E. Mäkinen, Constructing a binary tree efficiently from its traversals, Res. Report A-1998-5, University of Tampere, Finland, April 1998
63. J.-L. Loday, *Realization of the Stasheff polytope*, http://lanl.arXiv.org/abs/math/0212126, 2002
64. J.A. Anderson, *Discrete Mathematics with Combinatorics*, NJ: Prentice Hall, 2001

Chapter 2
Hierarchical Placement with Layout Constraints

Mark Po-Hung Lin and Yao-Wen Chang

Abstract In analog layout design, devices are required to be placed with matching, symmetry, and proximity constraints to reduce parasitic coupling effects and improve circuit performance. In addition to these basic placement constraints, there exist hierarchical symmetry and hierarchical proximity constraints due to circuit and layout design hierarchies. This chapter first introduces the hierarchical constraints induced by circuit and layout design hierarchies, and then presents a hierarchical placement approach to better consider these hierarchical constraints and effectively reduce the search space.

2.1 Introduction

According to [8, 11], the basic analog layout constraints include *common-centroid*, *symmetry*, and *proximity* constraints, illustrated in Fig. 2.1. The common-centroid constraint is usually applied to a subcircuit of a current mirror or a differential pair to reduce process-induced mismatches among the devices. The symmetry constraint is always required in the layout design of the whole differential subcircuit. It helps reduce the parasitic mismatches between two identical signal flows in the differential subcircuit. The proximity constraint is widely used in the subcircuit of a common device model or a certain circuit functionality. It helps form a connected placement of a subcircuit so that the subcircuit can share a connected substrate/well region or be surrounded by a common guard ring to reduce the layout area, the interconnecting wire length, and the substrate coupling effect. In particular, the placement outline of each subcircuit with the proximity constraint can be irregularly rectilinear for better area utilization. Figure 2.1c shows an example placement of two subcircuits, $\{E_1, E_2, E_3\}$ and $\{F_1, F_2, F_3\}$, with the proximity constraint.

M.P.-H. Lin (✉)
Department of Electrical Engineering, National Chung Cheng University, Chiayi, Taiwan
e-mail: marklin@ccu.edu.tw

H.E. Graeb (ed.), *Analog Layout Synthesis: A Survey of Topological Approaches*,
DOI 10.1007/978-1-4419-6932-3_2, © Springer Science+Business Media, LLC 2011

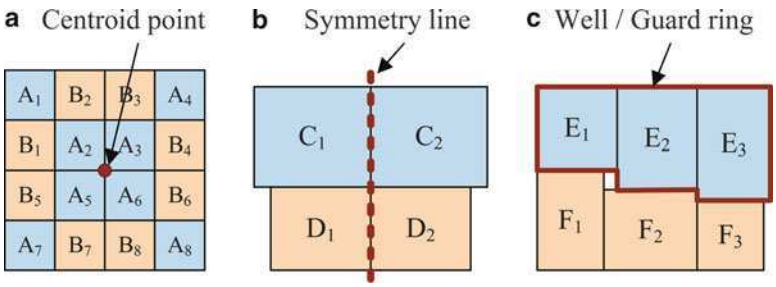


Fig. 2.1 Basic analog layout constraints (**a**) common-centroid constraint (**b**) symmetry constraint (**c**) proximity constraint

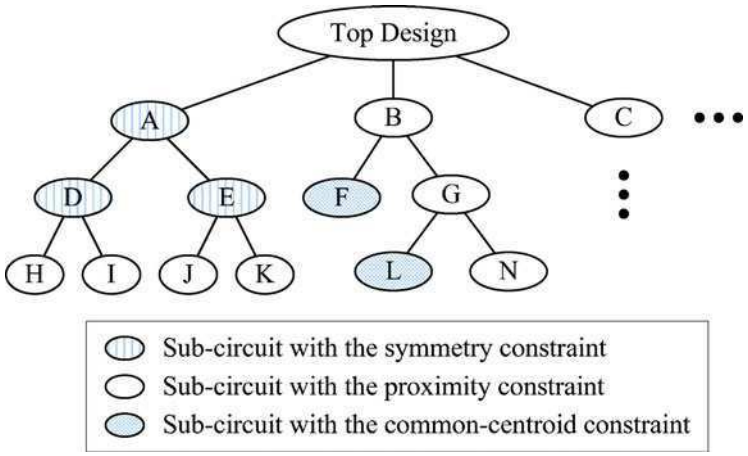


Fig. 2.2 Layout design hierarchy and the corresponding constraint in each subcircuit

Besides the basic layout constraints, there exist *hierarchical symmetry* and *hierarchical proximity* constraints due to *layout design hierarchy*. The layout design hierarchy may contain both exact and virtual hierarchies of analog circuit design. The exact hierarchy is the same as the circuit hierarchy, while the virtual hierarchy consists of hierarchical clusters [20]. Each cluster contains some devices and subcircuits, which are gathered based on device models, subcircuit functionality [10, 27], and/or other specific constraints [7]. Figure 2.2 shows an example layout design hierarchy, where each subcircuit corresponds to a specific constraint.

In Fig. 2.2, a subcircuit with the hierarchical symmetry constraint may contain some devices together with other subcircuits with the common-centroid and (hierarchical) symmetry constraints. Figure 2.3 shows an example hierarchical symmetric placement of several hierarchical subcircuits in Fig. 2.2. Similarly, a subcircuit with the hierarchical proximity constraint may contain some devices together with other subcircuits with the common-centroid, (hierarchical) symmetry, and (hierarchical) proximity constraints.

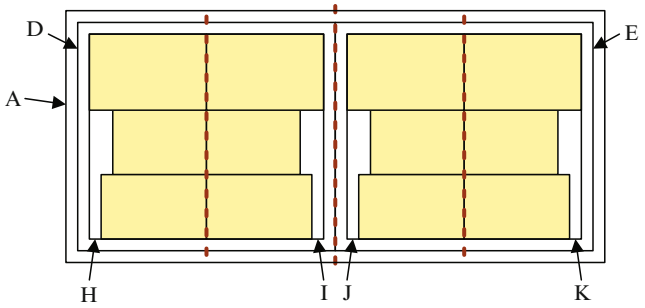


Fig. 2.3 An example placement of the subcircuits A, D, E, H, I, J, and K with the hierarchical symmetry constraint in Fig. 2.2, where the subcircuits H and I are symmetric, J and k are symmetric, and D and E are also symmetric

Based on the concept of the layout design hierarchy illustrated in Fig. 2.2, it is essential to synthesize analog layout hierarchically for better efficiency and effectiveness. It is also desirable to reduce the large search space by considering layout design hierarchy. Modern analog placement techniques often simultaneously optimize the placement in different hierarchical subcircuits, e.g., [17, 19, 20, 23, 24, 30], instead of bottom-up integration, because the optimal placement of a subcircuit may not lead to the globally optimal placement. Most of them apply simulated annealing [13] based on the topological floorplan representations, such as Sequence-Pair [28] and B*-tree [6], while the latest one [30] adopts a fully deterministic approach. Among these works, the one based on the hierarchical B*-tree (HB*-tree) in [19, 20, 23] discussed how to handle the hierarchical symmetry and hierarchical proximity constraints together with the consideration of layout design hierarchy.

In the following sections, the basic hierarchical framework based on the HB*-tree and symmetry-island formulation is first introduced to handle symmetry constraints. The generalized HB*-tree is then presented to consider both hierarchical symmetry and hierarchical proximity constraints.

2.2 Preliminaries

2.2.1 Symmetry Constraints

To reduce the effect of parasitic mismatches and circuit sensitivity to thermal gradients or process variations for analog circuits, some pairs of modules need to be placed symmetrically with respect to a common axis, and the symmetric modules are preferred to be placed at closest proximity for better electrical properties. The symmetry constraints can be formulated in terms of *symmetry types*, *symmetry groups*, *symmetry pairs*, and *self-symmetric modules*. In analog layout design, a symmetry group may contain some symmetry pairs and self-symmetric modules

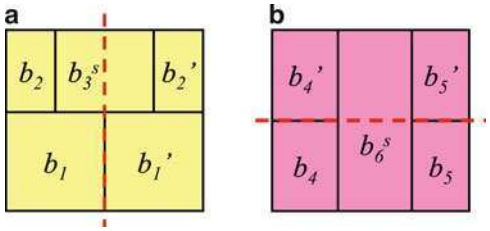


Fig. 2.4 Two symmetry types (**a**) symmetric placement with the vertical symmetry axis (**b**) symmetric placement with the horizontal symmetry axis

Table 2.1 The notations in this chapter

b	A module
S	A symmetry group
(b, b')	A symmetry pair
b^s	A self-symmetric module
b^r	The representative of a symmetry pair or a self-symmetric module
n	Number of modules
m	Number of symmetry groups
(x_i, y_i)	The center coordinate of the module b_i
w_i, h_i	The width and the height of the module b_i
$\hat{x}_i, \hat{y}_i$	The coordinate(s) of the symmetry axis (axes) of the symmetry group S_i

with respect to a certain symmetry type. A symmetry type may correspond to a symmetry axis in either the horizontal or the vertical direction. Figure 2.4 shows two different symmetry types with either the vertical or the horizontal symmetry axis.

For the symmetric placement with the vertical (horizontal) symmetry axis shown in Fig. 2.4a (Fig. 2.4b), a symmetry pair with two modules of the same dimensions and orientations should be placed symmetrically along the vertical (horizontal) symmetry axis. A self-symmetric module whose internal structure is self-symmetric must have its center placed at the symmetry axis.

The notations listed in Table 2.1 are used throughout this chapter. Let $S = \{S_1, S_2, \ldots, S_m\}$ be a set of m symmetry groups whose coordinate(s) of the symmetry axis (axes) is (are) denoted by $\hat{x}_i$ or $\hat{y}_i$ ($\hat{x}_i$ and $\hat{y}_i$), $1 \leq i \leq n$. A symmetry group $S_i = \{(b_1, b_1'), (b_2, b_2'), \ldots, (b_p, b_p'), b_1^s, b_2^s, \ldots, b_q^s\}$ consists of p symmetry pairs and q self-symmetric modules, where (b_j, b_j') denotes a symmetry pair and b_k^s denotes a self-symmetric module. Let (x_j, y_j) and (x_j', y_j') denote the respective coordinates of the centers of two modules b_j and b_j' in a symmetry pair (b_j, b_j'), and (x_k^s, y_k^s) denote the coordinate of the center of the self-symmetric module b_k^s. The symmetric placement of a symmetry group S_i with the vertical (horizontal) symmetry axis must satisfy (2.1) [(2.2)].

$$\begin{aligned} x_j + x_j' &= 2 \times \hat{x}_i, \quad && \forall j = 1, 2, \ldots, p. \\ y_j &= y_j', && \forall j = 1, 2, \ldots, p. \\ x^s{}_k &= \hat{x}_i, && \forall k = 1, 2, \ldots, q. \end{aligned} \tag{2.1}$$

$$\begin{aligned} x_j &= x'_j, && \forall j = 1, 2, \ldots, p. \\ y_j + y'_j &= 2 \times \hat{y}_i, && \forall j = 1, 2, \ldots, p. \\ y^s{}_k &= \hat{y}_i, && \forall k = 1, 2, \ldots, q. \end{aligned} \tag{2.2}$$

2.2.2 Symmetry Island

Before introducing the symmetry island, the effect of the symmetric device layout on the electrical matching properties of the symmetric devices should be investigated. Pelgrom et al. [29] measured the mismatch between MOS transistors with various electrical parameters as a function of device areas, distances, and orientations. According to [29], the difference of an electrical parameter P between two rectangular devices is modeled by the standard deviation as shown in (2.3), where A_P is the area proportionality constant for P, W and L denote the respective width and length of the device, and S_P denotes the variation of P under the device spacing D_x.

$$\sigma^2(\Delta P) = \frac{A_P^2}{WL} + S_P^2 D_x^2. \tag{2.3}$$

The device dimensions of modules in a symmetry pair are assumed to be the same. According to the above equation, the larger the distance between the symmetry pair, the greater differences between their electrical properties. Therefore, it is of significant importance for the symmetric devices of a symmetry group to be placed in close proximity. Figure 2.5a shows an analog circuit of a two-stage CMOS operational amplifier containing the differential input sub-circuit. The devices M1, M2, M3, M4, and M5 in the differential input sub-circuit form a symmetry group $S = \{(M1, M2), (M3, M4), M5\}$. Figures 2.5b, c show two corresponding layouts with different placement styles for the symmetry group S. The layout style in Fig. 2.5c is generally considered much better than that in Fig. 2.5b because the symmetric modules of the same symmetry group are placed at closer proximity (or even adjacent) to each other. Consequently, the sensitivities due to process variations can be minimized, and the circuit performance can be improved.

Based on the placement with the closest proximity for a symmetry group as shown in Fig. 2.5c, the concept of symmetry islands is then introduced with its definition given as follows:

Definition 2.1. A symmetry island is a placement of a symmetry group in which each module in the group abuts at least one of the other modules in the same group, and all modules in the symmetry group form a connected placement.

In the example of Fig. 2.6, the symmetry group S_1 in Fig. 2.6a forms a symmetry island, but that in Fig. 2.6b does not since it results in two disconnected components. The placement style in Fig. 2.6a is preferred in analog layout design due to its better electrical properties.

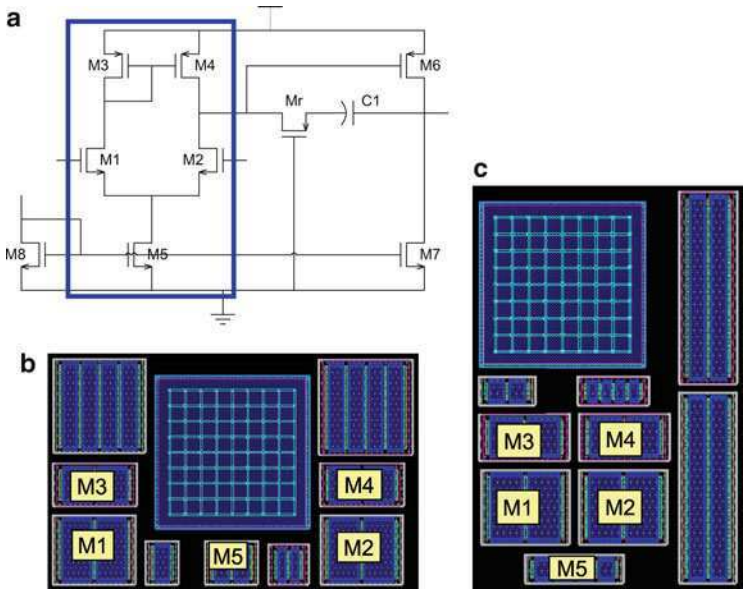


Fig. 2.5 An example analog circuit and two different layout styles for the circuit. (**a**) The schematic of a two-stage CMOS operational amplifier, where the differential input sub-circuit forms a symmetry group. (**b**) A layout design of the circuit in (**a**), where the devices of a symmetry group are not placed close to each other. (**c**) Another layout design of the circuit in (**a**), where the devices of a symmetry group are placed close to each other

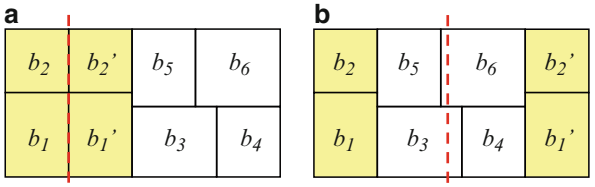


Fig. 2.6 Two symmetric-placement examples of a symmetry group $S_1 = \{(b_1, b_1'), (b_2, b_2')\}$. (**a**) S_1 forms a symmetry island. (**b**) S_1 cannot form a symmetry island

2.2.3 Review of B-Trees*

A B*-tree is an ordered binary tree representing a *compacted placement*, in which every module cannot move left and bottom anymore. As shown in Fig. 2.7, every node of a B*-tree corresponds to a module of a compacted placement. The root of a B*-tree corresponds to the module on the bottom-left corner. For each node n corresponding to a module b, the left child of n represents the lowest, adjacent module on the right side of b, while the right child of n represents the first module above b with the same horizontal coordinate.

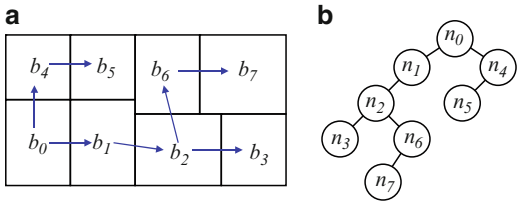


Fig. 2.7 (**a**) A compacted placement (same as Fig. 2.6a). (**b**) The B*-tree representing the compacted placement in (**a**)

Given a B*-tree, we can calculate the coordinate of each module by a preorder tree traversal. Suppose the module b_i, represented by the node n_i, has the bottom-left coordinate (x_i, y_i), the width w_i, and the height h_i. Then for the left child, n_j, of n_i, $x_j = x_i + w_i$; for the right child, n_k, of n_i, $x_k = x_i$. In addition, we maintain a contour structure to calculate the y-coordinates. Thus, starting from the root node, whose bottom-left coordinate is (0, 0), then visiting the root's left subtree, and then its right subtree, this preorder tree traversal procedure, a.k.a. B*-tree packing, calculates all coordinates of the modules in the placement. Using a doubly-linked list to implement the contour structure, the total packing time is linear to the number of modules.

2.3 Placement of a Symmetry Group

2.3.1 Automatically Symmetric-Feasible B-tree*

To consider the symmetric placement of a symmetry group and the packing of the symmetry modules to make a symmetry island, the automatically symmetric-feasible B*-tree (ASF-B*-tree for short) is proposed. Like B*-trees, the ASF-B*-tree can represent only *compacted* symmetric placement; in particular, there exists a unique correspondence between a compacted symmetric placement of a symmetry group and its induced ASF-B*-tree which results in a symmetry island.

Before introducing the ASF-B*-tree, the *representative* of a symmetry pair, the representative of a self-symmetric module, and the *representative B*-tree* are defined as follows:

Definition 2.2. The representative b_j^r of a symmetry pair (b_j, b'_j) is b'_j.

Definition 2.3. The representative b_k^r of a self-symmetric module b_k^s is the right (top) half of b_k^s in a symmetric placement with respect to a (horizontal) symmetry axis.

For the example of Fig. 2.8, the representative b_1^r of the symmetry pair $\{b_1, b'_1\}$ is b'_1, while the representative b_0^r of the self-symmetric module b_0^s is the right half of b_0^s.

It should be noted that each symmetry pair or self-symmetric module must have its own representative module. Therefore, the number of the representatives in a

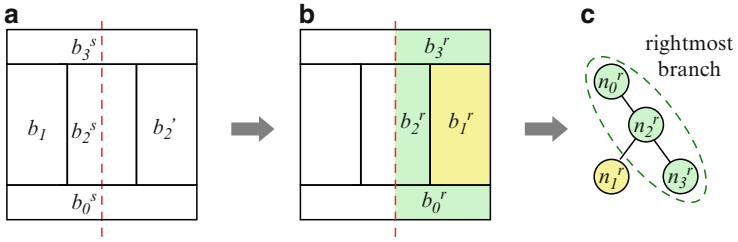


Fig. 2.8 (**a**) A placement example of a symmetry group have a vertical symmetry axis. (**b**) Selecting a representative for each symmetry pair and self-symmetric module. (**c**) The ASF-B*-tree (also a representative B*-tree) representing the placement of the symmetry group, where the *dash circled* nodes represent the left-boundary modules

symmetry group should be the same as the number of symmetry pairs and self-symmetric modules. The representative B*-tree is then defined as follows:

Definition 2.4. A representative B*-tree is a B*-tree containing only the representative nodes that correspond to representative modules.

Before explaining how to obtain an ASF-B*-tree by making a representative B*-tree symmetric-feasible for symmetric placements with vertical and horizontal symmetry axes, the *mirrored placement* of the representative modules for a symmetry group is introduced and defined as follows:

Definition 2.5. The mirrored placement of the representative modules for a symmetry group S_i is to place the nonrepresentative modules on the mirrored positions of the representative ones for each symmetry pair or each self-symmetric module in S_i with respect to its symmetry axis (axes). Furthermore, the representative and the nonrepresentative modules of each self-symmetric module are not disjointed.

Based on the definition of the mirrored placement of the representative modules, the symmetric-feasible condition of a representative B*-tree for the symmetric placements can be further defined as follows:

Definition 2.6. A representative B*-tree is symmetric-feasible if the mirrored placement of the representative modules can be obtained after packing the representative B*-tree.

In Fig. 2.8a, the modules in the symmetry group $S = \{(b_1, b_1'), b_0^s, b_2^s, b_3^s\}$ are placed symmetrically with respect to the vertical axis. To construct the corresponding representative B*-tree, the representative module of each symmetry pair and self-symmetric module should be selected with the consideration of the placement on the right-half plane. Figure 2.8b highlights the representative modules, and Fig. 2.8c shows the corresponding representative B*-tree of the symmetric placement. Each node in the representative B*-tree corresponds to a representative module.

To make the representative B*-tree symmetric-feasible, the following lemmas are derived, which formulate the symmetry conditions for self-symmetric modules and symmetry pairs.

Lemma 2.1. *The representative of a self-symmetric module must abut the symmetry axis.*

Proof. Let S be a symmetry group with a vertical symmetry axis, and b^s be a self-symmetric module in S. The symmetry axis of S is denoted by $\hat{x}$, and the center of b^s is denoted by (x^s, y^s).

Based on (2.1), the symmetry axis $\hat{x}$ always passes through the center (x^s, y^s) of the self-symmetric module b^s, i.e., $\hat{x} = x^s$. According to Definition 2.3, the representative b^r of b^s is the right half of b^s. Therefore, the center (x^s, y^s) of b^s must be on the left boundary of b^r. To keep the symmetric-feasible condition $\hat{x} = x^s$, b^r must abut the symmetry axis $\hat{x}$. The case for a symmetry group with a horizontal symmetry axis can be proved similarly.

Lemma 2.2. *The representative of a symmetry pair not on a symmetry axis is always symmetric-feasible.*

Proof. Let S be a symmetry group with a vertical symmetry axis, and (b, b') be a symmetry pair in S. The symmetry axis of S is denoted by $\hat{x}$. The respective centers of b and b' are (x, y) and (x', y'), and the respective widths/heights of b and b' are w/h and w'/h', where $w = w'$ and $h = h'$. The representative of the symmetry pair (b, b') is b'.

Given the coordinate of the representative b' and the vertical symmetry axis $\hat{x}$, the coordinate of the symmetric module b can be calculated by (2.1). We have $x = 2 \times \hat{x} - x'$ and $y = y'$. After transposing $\hat{x}$ to the left side and having the absolute value on both sides, we have $|x - \hat{x}| = |\hat{x} - x'|$. Since the representative is not on the symmetry axis, we have $|x - \hat{x}| = |\hat{x} - x'| \geq \frac{w}{2}$. It means that the distances from the symmetry axis to the centers of b and b' are greater than or equal to half of the width of b or b'. Since b and b' are on different sides of the symmetry axis, b and b' will not overlap each other. Therefore, the symmetric-feasible condition is always satisfied. The case for a symmetry group with a horizontal symmetry axis can be proved similarly.

According to Lemma 2.1 and the boundary constraints [21] in the B*-trees, the symmetric-feasible representative B*-trees have the following property:

Property 2.1. The left-boundary (right-boundary) constraint for the symmetric placement with respect to a vertical (horizontal) symmetry axis: the representative node of a self-symmetric module should always be on the right (left) most branch of the representative B*-tree.

Based on the above property, the nodes representing the modules on the left boundary should be on the rightmost branch as shown in Fig. 2.8c.

Similarly, the symmetric-feasible representative B*-tree of the symmetric placement when the symmetry axis is in the horizontal direction can be derived. In this

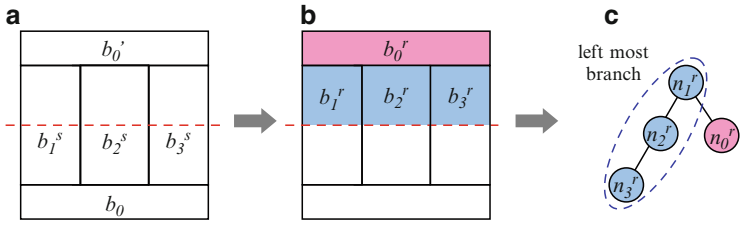


Fig. 2.9 (**a**) A placement example of a symmetry group with a horizontal symmetry axis. (**b**) Selecting a representative module for each symmetry pair and self-symmetric module. (**c**) The ASF-B*-tree (also a representative B*-tree) representing the placement of the symmetry group, where the *dash circled* nodes represent the bottom-boundary modules

case, we only consider the top-half plane during the placement of the representative modules. Figure 2.9c shows the representative B*-tree of the symmetry group $S = \{(b_0, b_0'), b_1^s, b_2^s, b_3^s\}$ having the symmetric placement with respect to the horizontal symmetry axis in Fig. 2.9a. Again, the representatives of the self-symmetric modules should abut the horizontal symmetry axis, which is on the bottom boundary of the top-half plane. Therefore, the nodes representing the modules on the bottom boundary should be on the leftmost branch, as illustrated in Fig. 2.9c.

Based on Definition 2.4 and Property 2.1, an ASF-B*-tree is defined as follows:

Definition 2.7. An ASF-B*-tree is a representative B*-tree, which satisfies Property 2.1.

Once an ASF-B*-tree is packed, the coordinates of these representatives are obtained, and the coordinates of their symmetric modules can be further calculated based on (2.1) and (2.2) with the given coordinates of the symmetry axes, $\hat{x}_i$ and $\hat{y}_i$. The symmetric placement of a symmetry group automatically forms a symmetry island.

Based on Lemmas 2.1 and 2.2, the following theorems are derived:

Theorem 2.1. *An ASF-B*-tree is symmetric-feasible in a symmetric placement of a symmetry group with respect to either a vertical or a horizontal symmetry axis.*

Proof. An ASF-B*-tree is symmetric-feasible if all the representatives in the ASF-B*-tree are symmetric-feasible. There are four kinds of representatives, and the symmetric-feasible condition for each is defined and proved in Lemmas 2.1 and 2.2. Therefore, an ASF-B*-tree is symmetric-feasible in a symmetric placement of a symmetry group with respect to either a vertical or a horizontal symmetry axis.

Theorem 2.2. *The packing of an ASF-B*-tree results in a symmetry island of the corresponding symmetry group.*

Proof. It is obvious that all the representative modules will form a connected placement after packing. We set the coordinate(s) of the symmetry axis (axes) to the left or (and) the bottom boundary (boundaries) of the connected placement

of the representative modules. The coordinates of the symmetric modules can be calculated by (2.1) and (2.2). The symmetric modules also form a connected placement, and the boundary of the connected placement also abut the symmetry axis (axes). Therefore, the whole symmetry group form a connected placement, and each module in the group abuts at least one of the other modules in the same group. The packing of an ASF-B*-tree thus results in a symmetry island of the corresponding symmetry group.

Theorem 2.3. *There exists a unique correspondence between a compacted symmetric placement of a symmetry group and its induced ASF-B*-tree.*

Proof. According to [6], there is a unique correspondence between an admissible placement and its induced B*-tree. After obtaining the placement of the representative modules, the mirrored placement of the symmetric ones is also obtained. The mirrored placement is also unique. Therefore, there exists a unique correspondence between a compacted symmetric placement of a symmetry group and its induced ASF-B*-tree.

Based on the above theorems, a corresponding symmetric placement for an ASF-B*-tree can correctly and efficiently be found by avoiding searching in redundant solution spaces. It will be clear later in Sect. 2.6 that these advantageous properties of ASF-B*-trees lead to superior solution quality and efficiency for analog placement.

2.3.2 ASF-B*-Tree Packing

The packing of the ASF-B*-tree is similar to that of the B*-tree [6], which follows the preorder tree traversal procedure to calculate the coordinates of the modules. During the packing, two double linked lists are implemented to keep both horizontal and vertical contour structures. Figure 2.10 shows the packing procedure of the example ASF-B*-tree in Fig. 2.13a. The bold (red) lines denote the horizontal contour, while the dotted (green) lines represent the vertical contour.

After obtaining the coordinates of all representative modules in the symmetry group, the coordinates of the symmetric modules and the extended contours can be

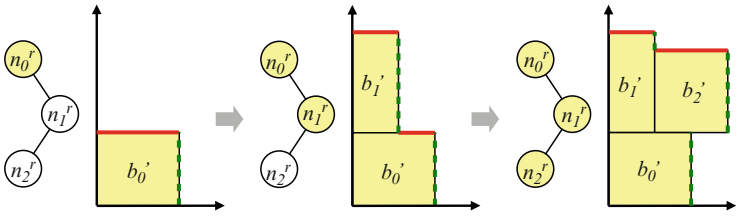


Fig. 2.10 The packing procedure including the contour updates of the ASF-B*-tree in Fig. 2.13a

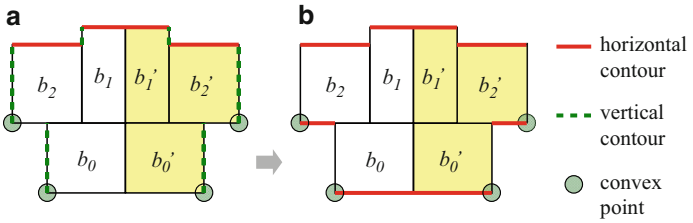


Fig. 2.11 The generation of the bottom contour of the symmetry island based on the dual vertical contours. (**a**) The convex points obtained by traversing the dual vertical contours from *bottom* to *top*. (**b**) The *bottom* horizontal contour connected by the convex points

calculated based on either (2.1) or (2.2). Figure 2.13b shows the resulting placement of the symmetry group and the contours of the symmetry island for the ASF-B*-tree shown in Fig. 2.13a. As shown in Fig. 2.13b, the symmetry island contains one top horizontal and dual vertical contours. To further calculate the bottom horizontal contour of the symmetry island, both vertical contours from bottom need to be traversed to top and keep the convex points as shown in Fig. 2.11a. By connecting the convex points horizontally, the bottom horizontal contour of the symmetry island can be obtained as shown in Fig. 2.11b.

2.4 The Hierarchical Framework

2.4.1 Hierarchical HB-Tree*

The hierarchical framework, called *hierarchical B*-tree* (*HB*-tree* for short), is proposed to handle the simultaneous placement of modules in symmetry islands and nonsymmetric modules. In an HB*-tree, the symmetry island of each symmetry group can be in any rectilinear shapes, and symmetry and nonsymmetric modules are simultaneously placed to optimize the placement.

Figure 2.12 shows an HB*-tree for the placement in Fig. 2.6a. Two symmetry groups, S_1 and S_2, are represented by two hierarchy nodes, n_{S_1} and n_{S_2}, and each hierarchy node contains an ASF-B*-tree that corresponds to a symmetry island in the symmetric placement.

2.4.2 HB-Tree with Rectilinear Symmetry Islands*

The symmetry islands are often not rectangular, but are of rectilinear shapes. For example, in Fig. 2.13c, the symmetry island of the symmetry group S_0 is of the rectilinear shape. Therefore, the HB*-tree in Fig. 2.12 should be augmented to handle rectilinear symmetry islands. Wu et al. [33] proposed a method to deal with

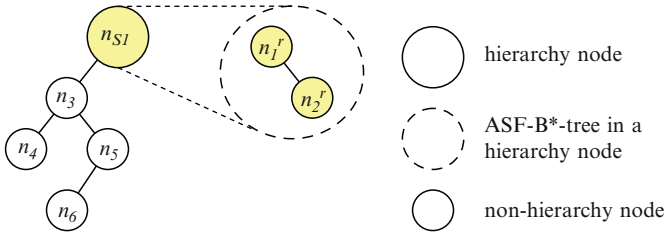


Fig. 2.12 An HB*-tree for the placement in Fig. 2.6a

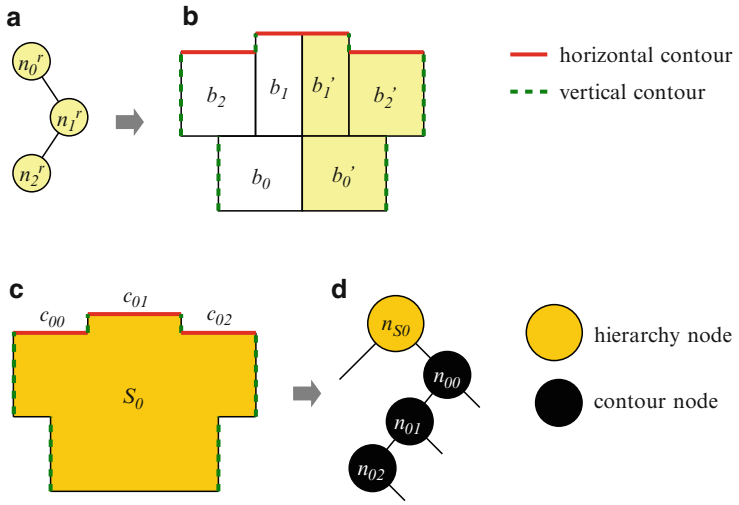


Fig. 2.13 (**a**) An ASF-B*-tree of a symmetry group S_0. (**b**) The horizontal and vertical contours of the corresponding placement. (**c**) The symmetry island and its effective contours. (**d**) The HB*-tree for the rectilinear symmetry island

rectilinear modules by slicing a rectilinear module into several rectangular submodules along each vertical boundary. However, it is complicated to maintain the relationship between the submodules during B*-tree perturbations.

Instead of slicing a rectilinear symmetry island, several *contour nodes* are introduced to represent top horizontal contour segments of the symmetry island. In Fig. 2.13c, there are three horizontal contour segments, c_{00}, c_{01}, and c_{02}. The HB*-tree is augmented by introducing the three contour nodes, n_{00}, n_{01}, n_{02}, as shown in Fig. 2.13d. Each contour node keeps the coordinates of the corresponding horizontal contour segment. The relationship of a hierarchy node, its contour nodes, and other regular module nodes is described as follows:

Property 2.2. Properties for an HB*-tree.

1. The left child of a hierarchy node, if any, must be a noncontour node.
2. The right child of a hierarchy node must be the contour node representing the leftmost top horizontal contour segment of the symmetry island.

3. The left child of a contour node, if any, must be the contour node representing the next contour segment on the right side.
4. The children of a regular module node must be a noncontour node.
5. The right child of a contour node, if any, must be a noncontour node.
6. The parent of a contour node cannot be a regular module node.

Proof. Given a symmetry group S_0, b_{S_0} denotes the symmetry island of S_0, n_{S_0} denotes the corresponding hierarchy node, and n_{0i} represents the ith top contour segment of b_{S_0} from left to right.

1. Since the contour node n_{0i} represents the ith top contour segment of b_{S_0}, it is impossible for n_{0i} to be the left child of n_{S_0} that corresponds to the lowest, adjacent module on the right side of b_{S_0}, based on the B*-tree definition. The property thus follows.
2. According to the definition of the B*-tree, the right child of n_{S_0} represents the first module above b_{S_0}. Since the top horizontal contour segments of b_{S_0} always abut b_{S_0}, other modules cannot be placed between b_{S_0} and its top contour segments. Therefore, the right child of n_{S_0} must be a contour node representing the leftmost top horizontal contour segment of b_{S_0}.
3. By the contour node definition, the contour node $n_{0,i}$ represents the ith top contour segment of b_{S_0} from left to right, and the left child of $n_{0,i}$, if any, is $n_{0,i+1}$ representing the next $((i+1)$th) contour segments. If $n_{0,i}$ represents the last (the rightmost) top contour segment, the left child of n_{0i} is empty.
4. Based on the second and the third properties of the HB*-tree, the contour node n_{0i} cannot be the left or right child of a regular module node. The property thus follows.
5. The right child of the contour node n_{0i} represents the first module above the ith top contour segment of b_{S_0}. If there exists another contour node n_{0j} that is the right child of n_{0i}, both contour segments will overlap each other with n_{0j}'s contour segment on top of that of n_{0i}, implying that n_{0i} is not a contour node. This is a contradiction.
6. Based on the construction of the HB*-tree, the parent of a contour node is either a contour node or a hierarchy node.

Figure 2.13a shows the ASF-B*-tree of the symmetry group $S_0 = \{(b_0, b'_0), (b_1, b'_1), (b_2, b'_2)\}$. In Fig. 2.13b, the horizontal and vertical contours are obtained from the rectilinear outline after packing the ASF-B*-tree. Figure 2.13c shows the symmetry island and the effective horizontal and vertical contours. The horizontal contour segments are denoted as c_{00}, c_{01}, and c_{02} from left to right. Therefore, we have a hierarchy node n_{S_0} representing the symmetry island of the symmetry group S_0, and three contour nodes n_{00}, n_{01}, and n_{02} representing the contour segments. The relationship between the hierarchy node and its contour nodes is shown in the HB*-tree in Fig. 2.13d.

2.4.3 HB*-Tree Packing

The HB*-tree packing also adopts the preorder tree traversal procedure. When a hierarchy node is traversed, the ASF-B*-tree in the hierarchy node should be packed first to obtain the contours of the symmetry island described previously. The contours are then stored in the corresponding hierarchy node. During packing a hierarchy node representing a symmetry island, the best packing coordinate for the bottom boundary of the symmetry island should be calculated based on the bottom contour shown in Fig. 2.11b. The left child of the hierarchy node is then proceeded to be packed. After the left child and all its descendants are packed, the first contour node of the symmetry island is packed, followed by the second one, and so on. When packing the contour nodes, their corresponding coordinates should be updated and the hierarchy node should be replaced in the contour data structure of the HB*-tree.

Figure 2.14a shows an HB*-tree representing 20 modules with two symmetry groups S_0 and S_1. For the packing, the two ASF-B*-trees in n_{S_0} and n_{S_1} are packed first, and the rectilinear outlines of the two symmetry islands are obtained. Then, the nodes, n_5, n_6, n_7, n_8, n_9, are packed in the DFS order. The temporal contour list

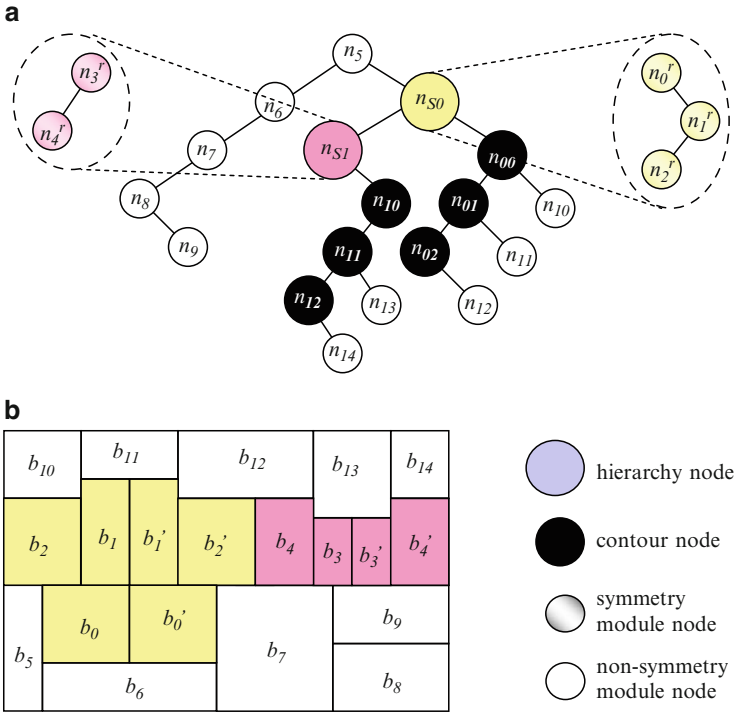


Fig. 2.14 (**a**) An HB*-tree representing 20 modules with two symmetry groups S_0 and S_1. (**b**) The resulting placement after packing the HB*-tree

is $< n_5, n_6, n_7, n_9 >$. By calculating the rectilinear outlines between the temporal contour list and the bottom boundary of the symmetry island S_0, the dead space between the previously packed modules and the symmetry island can be minimized. The updated temporal contour list becomes $< n_{S_0}, n_7, n_9 >$. Continuing the packing procedure, the resulting placement of the HB*-tree is obtained as shown in Fig. 2.14b finally. Although the purpose of the packing is to obtain a compacted placement, sufficient white space might need to be allocated for the surrounding wells or guard rings based on the device types, such as NMOS or PMOS transistors. When packing a node, the device type of the corresponding module should be compared with those of the previously packed modules in the current contour list. If the device types are different, the currently packed module should be snapped to a position to reserve sufficient white space for the surrounding wells or guard rings.

The following theorem shows the packing complexity.

Theorem 2.4. *The packing for an ASF-B*-tree or an HB*-tree takes linear time.*

Proof. Given a design with n modules (including symmetry and nonsymmetry ones) and m symmetry groups, let $\hat{n}$ be the number of nonsymmetric modules and $n(S_i)$ be the number of modules in each symmetry group S_i, where $n(S_i) \geq 1$. We have $n = \hat{n} + \sum_{i=1}^{m} n(S_i)$.

For the HB*-tree representing the symmetric placement of the given design, there are m hierarchy nodes, $O(\sum_{i=1}^{m} n(S_i))$ contour nodes, and $\hat{n}$ module nodes. For the ASF-B*-tree of the symmetry group S_i in a hierarchy node, there are $O(n(S_i))$ representative nodes.

First, the packing for the ASF-B*-tree of the symmetry group S_i in a hierarchy node is considered. It consists of two steps. The first step is the packing for all representative modules. The second step is the calculation of the coordinate of each symmetric module.

According to [6], the packing for a B*-trees takes linear time, so the time complexity of the first step is $O(n(S_i))$. Since it takes constant time to calculate the coordinate of a symmetric module, it also takes $O(n(S_i))$ time to compute the coordinates of all the symmetric modules in S_i. Combining both steps, we have the $O(n(S_i))$ time complexity for the packing of an ASF-B*-tree of S_i.

Second, the packing for the HB*-tree is considered. If all the symmetry islands of m symmetry groups are in a rectangular shape, we can ignore the contour nodes in the HB*-tree, and it takes $O(m + \hat{n})$ time to pack the HB*-tree. However, if any symmetry island is in a rectilinear shape, we need to consider the packing of the hierarchy node representing this symmetry island, especially the additional contour nodes.

The bottom contour of the symmetry island of S_i is obtained when the corresponding ASF-B*-tree of the symmetry group is packed, and the number of the bottom contour segments is $O(n(S_i))$. By comparing the current packing contour segments and the bottom contour segments of the symmetry island from left to right, it also takes $O(n(S_i))$ time to get the coordinates of the modules in the symmetry island S_i.

To sum up, it takes $O(m + \sum_{i=1}^{m} n(S_i) + \hat{n})$ time to pack the HB*-tree. Since $n = \sum_{i=1}^{m} n(S_i) + \hat{n}$, the packing time can be reduced to $O(m+n)$ time. Since the number of symmetry group m is upper bounded by the number of total modules n, the packing time is $O(n)$.

2.5 The Algorithm

The placement algorithm is based on simulated annealing [13]. Given a set of modules and symmetry constraints as the inputs, an initial solution represented by an HB*-tree is constructed and then perturbed to search for a desired configuration until a predefined termination condition is satisfied. The cost function, $\Phi(P)$, of the placement is defined in (2.4), where α and β are user-specified parameters, A_P is the area of the bounding rectangle for the placement, and W_P is the half-perimeter wire length (HPWL).

$$\Phi(P) = \alpha \times A_P + \beta \times W_P. \tag{2.4}$$

2.5.1 HB-Tree Perturbation*

The following operations are applied to perturb an HB*-tree.

- Op1: Rotate a module.
- Op2: Move a node to another place.
- Op3: Swap two nodes.

In the perturbation, the nonhierarchy nodes have higher probabilities to be selected because rotating, moving, or swapping the hierarchy nodes might incur a big jump in finding the next solution. It is well known that such a big jump might deteriorate the solution quality during the SA process. It should be noted that, due to the special structure of the HB*-tree, a non-hierarchy node cannot be moved to the right child of a hierarchy node or the left child of a contour node. The contour nodes are always moved along with its hierarchy node which cannot be moved individually.

2.5.2 ASF-B-Tree Perturbation*

In addition to the aforementioned Op1, Op2, and Op3 for HB*-tree perturbation, the operations, Op4 and Op5, are introduced to perturb the ASF-B*-trees. It should be noted that Property 2.1 should always be satisfied when perturbing an ASF-B*-tree according to the definition of the ASF-B*-trees in Definition 2.7.

- Op4: Change a representative.
- Op5: Convert a symmetry type.

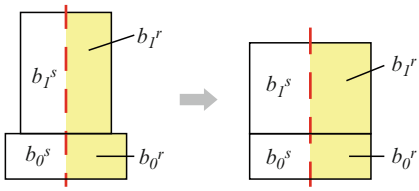


Fig. 2.15 Rotating the self-symmetric module b_1^s in the symmetry group $S = \{b_0^s, b_1^s\}$ results in the shape change of its representative b_1^r

2.5.2.1 Module Rotation

When rotating modules in a symmetry group, the corresponding ASF-B*-tree is unchanged. Two cases of symmetry-module rotation should be considered.

- Case 1: Rotate a symmetry pair.
- Case 2: Rotate a self-symmetric module.

In Case 1, both modules of a symmetry pair should be rotated at the same time so that they can still be symmetrically placed with respect to a symmetry axis. In Case 2, after rotating a self-symmetric module, the shape of its representative should be updated accordingly as shown in Fig. 2.15.

2.5.2.2 Node Movement

When moving a node to another place in an ASF-B*-tree, the following two cases should be considered.

- Case 1: Move a node representing the representative of a symmetry pair.
- Case 2: Move a node representing the representative of a self-symmetric module.

In Case 1, the representative node of a symmetry pair can be moved to anywhere in an ASF-B*-tree. In Case 2, however, the representative node of a self-symmetric module can only be moved along the rightmost (leftmost) branch of the ASF-B*-tree for vertical (horizontal) symmetric placement so that Property 2.1 is satisfied.

2.5.2.3 Node Swapping

When swapping two nodes in an ASF-B*-tree, the following two cases should be considered.

- Case 1: Both nodes represent the representatives of two different symmetry pairs.
- Case 2: At least one node represents the representative of a self-symmetric module.

In Case 1, two nodes representing the representatives of two different symmetry pairs can be arbitrarily swapped. However, for Case 2, if at least one of the swapped

nodes represents the representative of a self-symmetric module, the other node must be located on the same branch (i.e., the leftmost or the rightmost branch) of the ASF-B*-tree. Therefore, Property 2.1 is still satisfied after node swapping.

2.5.2.4 Representative Change

The purpose of changing a representative for a symmetry pair or a self-symmetric module is to optimize the wire length, while the area is kept unchanged after changing the representative. The representative of either a symmetry pair or a self-symmetric module can be changed.

- Case 1: Change the representative of a symmetry pair.
- Case 2: Change the representative of a self-symmetric module.

In Case 1, for a symmetry pair (b_j, b'_j), the representative can be changed from b_j to b'_j or from b'_j to b_j. Figure 2.16 illustrates that changing the representative of the symmetry pair (b_1, b'_1) from b'_1 to b_1 may result in shorter wire length between b_1 and b_3. Similarly, in Case 2, for a self-symmetric module b^s_k, we can change its representative by flipping it horizontally or vertically according to its symmetry axis. As illustrated in Fig. 2.17, changing the representative of the self-symmetric module b^s_1 by flipping it horizontally may result in shorter wire length between b^s_1 and b_3. Obviously, each operation takes constant time.

2.5.2.5 Symmetry-Type Conversion

For symmetry-type conversion of a symmetry group, both conversions between the vertical symmetry and the horizontal one should be considered.

- Case 1: Convert the symmetry type from vertical symmetry to horizontal one.
- Case 2: Convert the symmetry type from horizontal symmetry to vertical one.

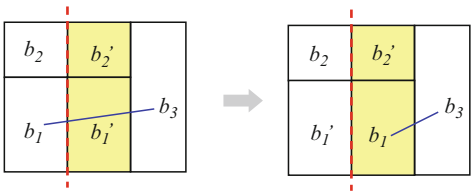


Fig. 2.16 Changing the representative of the symmetry pair (b_1, b'_1) from b'_1 to b_1 may result in shorter wire length between b_1 and b_3

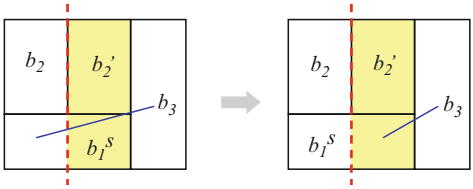


Fig. 2.17 Changing the representative of the self-symmetric module b^s_1 may result in shorter wire length between b^s_1 and b_3

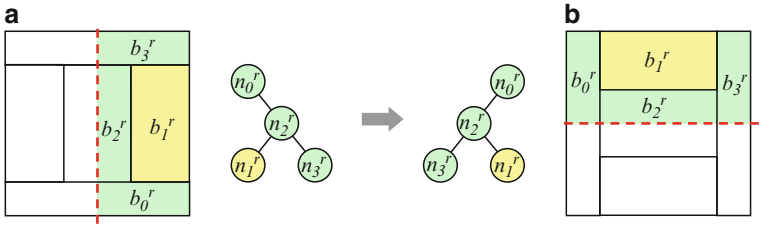


Fig. 2.18 Converting the symmetry type from (**a**) vertical symmetry to (**b**) horizontal symmetry

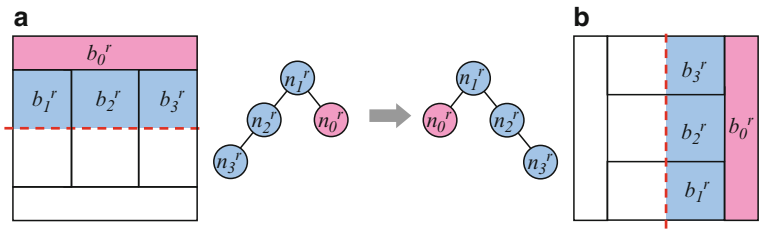


Fig. 2.19 Converting the symmetry type from (**a**) horizontal symmetry to (**b**) vertical symmetry

To convert the symmetry type of a symmetry group from vertical symmetry to horizontal one or vice versa, we first rotate every module including the representative, and then swap the left and the right children of each node in the given ASF-B*-tree. Figures 2.18 and 2.19 show the respective examples for the conversions of Cases 1 and 2.

It should be noted that the symmetry type is usually predefined based on the power/ground lines or signal flows in the layout by the analog designers. Therefore, Op5 is seldom applied in real applications.

2.5.3 Contour Node Related Updates

Once an ASF-B*-tree is perturbed, the number of the corresponding contour nodes in the HB*-tree might be changed. The tree structure might have to be updated accordingly. If the number of contour nodes representing the horizontal contour segments of the symmetry island is increased, the structure of the HB*-tree can be kept unchanged. However, if that of the contour nodes is decreased, some other nodes in the HB*-tree might not have parents. Such nodes, called *dangling nodes* should be reassigned to new parents. To keep the relative placement topology before and after perturbing an ASF-B*-tree, the nearest contour node is searched for each dangling node. If the nearest contour node has no right child, it is the parent of the dangling node, and the dangling node will be its right child. If the nearest contour node has a right child, the leftmost-skewed child of the right child is traversed.

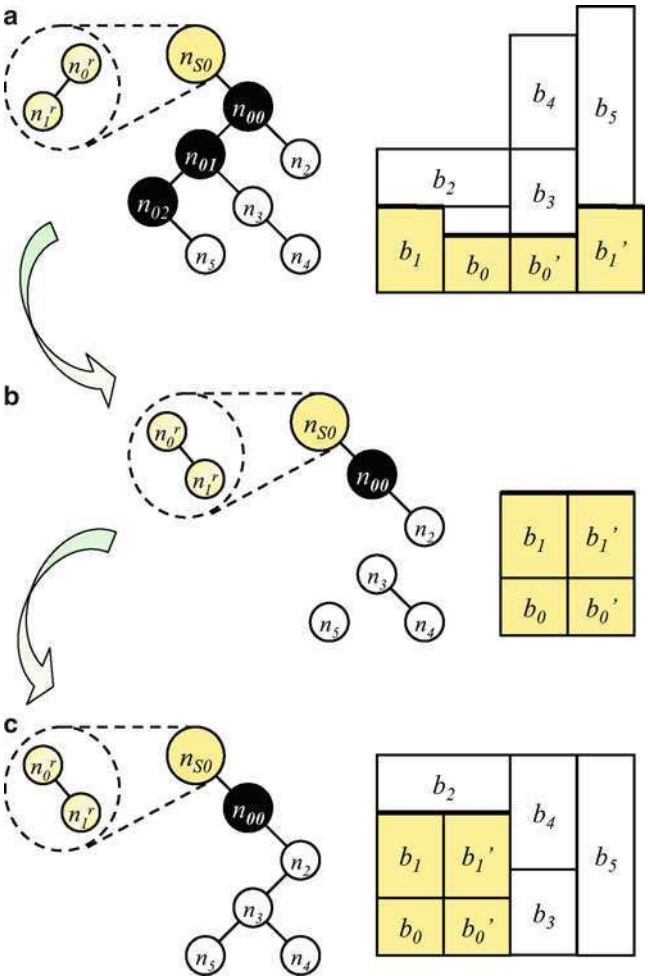


Fig. 2.20 An example of updating contour-related nodes. (**a**) An HB*-tree and its corresponding placement containing the symmetry group $S_0 = \{(b_0, b_0'), (b_1, b_1')\}$. (**b**) The intermediate HB*-tree after perturbing the ASF-B*-tree in the hierarchy node n_{S0}, and the corresponding symmetry island of S_0. The contour-related nodes, n_3 and n_5, become dangling. (**c**) The HB*-tree after updating the contour-related nodes and its corresponding placement

The leftmost-skewed child will be the parent of the dangling node, and the dangling node is assigned to its left child. It takes amortized constant time to update the contour related nodes.

Figure 2.20 shows an example of updating contour-related nodes. In Fig. 2.20a, there are initially three contour nodes representing the three top contour segments of the symmetry island of the symmetry group S_0. After performing Op2 to perturb the ASF-B*-tree in n_{S_0}, the representative node n_1^r is moved from the left child to the right child of the other representative node n_0^r. The placement of S_0 forms

a new symmetry island as shown in Fig. 2.20b, which has only one top contour segment. Therefore, the contour nodes n_{01} and n_{02} disappear, and the nodes n_3 and n_5 become dangling nodes. We first find the nearest contour node of n_3, which is n_{00}. Since n_{00} already has the right child n_2, the leftmost skewed child of n_2 should be searched. In this case, we directly assign n_3 to be the left child of n_2 because n_2 has no left child. After n_3 is assigned to a proper tree location, the nearest contour node of n_5 is then searched, which is also n_{00}. Since n_{00} already has the right child n_2, the leftmost skewed child is searched, which is n_3. Finally, n_3 is assigned to be the parent of n_5, and n_5 is assigned as the left child of n_3.

2.6 Comparisons with Other Approaches

In this section, we compare existing topological analog placement methods considering symmetry constraints, based on theoretical and empirical studies. The first subsection explores the time complexities of the perturbation and packing operations adopted by the existing topological methods, and the second subsection conducts experiments based on the simulated annealing algorithm and two sets of commonly used benchmarks.

2.6.1 Comparisons of Time Complexities

The problem of analog placement considering symmetry constraints has been extensively studied in the literature. Most of these works used the simulated annealing (SA) algorithm [13] in combination with floorplan representations to handle symmetry constraints. These representations can be classified into two major categories: (1) the absolute representation and (2) the topological representation.

For the absolute representation first proposed by Jepsen and Gellat [12], each module is associated with an absolute coordinate on a gridless plane. It operates on a module by changing its coordinate directly. The KOAN/ANAGRAM II [9], PUPPY-A [25], and LAYLA [16] systems all adopted the absolute representation to handle the placement of analog modules. The main weakness of the absolute method lies in the fact that it may generate an infeasible placement with overlapped modules. Therefore, a postprocessing step must be performed to eliminate this condition, which implies a longer computation time.

Recently, most previous works apply topological floorplan representations due to its flexibility and effectiveness. The most popular floorplan representations include the B*-tree [6], Sequence Pair (SP) [28], and TCG [18].

For the B*-tree representation, Balasa et al. derived its symmetric-feasible condition [1]. To explore the solution space in the symmetric-feasible B*-trees, they augmented the B*-tree [6] using various data structures, including segment trees [3, 5], red–black trees [4], and deterministic skip lists [26], to reduce the packing time.

Table 2.2 Comparisons of popular analog placement approaches considering symmetry constraints based on topological floorplan representations. n: the number of modules; m: the number of symmetry pairs

Analog placement approach considering symmetry constraints	Perturbation time	Packing time
B*-tree [1]	$O(\lg n)$	$O(n^2)$
B*-tree + Seg. tree [3]	$O(\lg n)$	$O(n \lg n)$
BT + RB-tree [4]	$O(\lg n)$	$O(n \lg n)$
BT + Skip list [26]	$O(\lg n)$	$O(n \lg n)$
Sequence-pair (SP) [2]	$O(1)$	$O(n^2)$
SP + LP [14]	$O(1)$	$\Omega(n^2)$
SP w. dummy [31]	$O(1)$	$O(n^2)$
SP w. priority queue [15]	$O(1)$	$O(m \cdot n \lg \lg n)$
TCG-S [22]	$O(n^2)$	$O(n^2)$
TCG [34]	$O(n)$	$O(n^2)$
B*-tree w. ESF + LP [30]	N/A	$\Omega(n^2)$
ASF-B*-tree + HB*-tree	$O(\lg n)$	$O(n)$

More recently, Strasser et al. [30] proposed a deterministic approach based on the B*-tree representation [6] with enhanced shape functions (ESF) and linear programming (LP).

For the SP representation, Balasa et al. also derived its symmetric-feasible condition [2]. By taking advantage of the symmetry-feasible condition, Koda et al. [14] proposed a linear programming based method. Tam et al. [31] introduced a dummy node and additional constraint edges for each symmetry group after obtaining a symmetric-feasible sequence pair. Krishamoorthy et al. [15] proposed an $O(m \cdot n \lg \lg n)$ packing-time algorithm by employing the priority queue, where m is the number of symmetry groups and n is the number of modules.

For the TCG representation, Lin et al. presented its symmetric-feasible conditions [22]. However, it requires $O(n^2)$ time to perturb and pack TCGs. Zhang et al. [34] further improved the perturbation time of the TCG representation from $O(n^2)$ to $O(n)$.

Table 2.2 compares aforementioned analog placement approaches considering symmetry constraints based on topological floorplan representations. It should be noted that "ASF-B*-tree + HB*-tree" is the fastest algorithm among all these popular approaches, while "SP + LP [14]" and "B*-tree w. ESF + LP [30]" take at least $\Omega(n^2)$ packing time due to LP. Since the approach [30] explores all placement configurations for a small set of device modules and groups in each hierarchy, the perturbation of the B*-tree is not required.

2.6.2 *Comparisons of Experimental Results*

The placement algorithms were implemented in the C++ programming language on a 3.2GHz Intel Pentium4 PC under the Linux operation system. Two sets of

Table 2.3 MCNC benchmark circuits

Circuit	# of mod.	# of sym. mod.	Mod. area (mm^2)
apte	9	8	46.56
hp	11	8	8.83
ami33	33	6	1.16
ami49	49	4	35.45

Table 2.4 Industry benchmark circuits

Circuit	# of mod.	# of sym. mod.	Mod. area ($10^3\ \mu m^2$)
biasynth_2p4g	65	$8 + 12 + 5$	4.70
lnamixbias_2p4g	110	$16 + 6 + 6 + 12 + 4$	46.00

experiments were performed: one is based on the four MCNC benchmarks (apte, hp, ami33, and ami49) used in [22], and the other consists of two real industry analog designs (biasynth_2p4g and lnamixbias_2p4g) used in [5] and [14]. (Note that they both were extracted by Koda et al. [14] from Figs. 9 and 10 in [5].) Table 2.3 lists the names of the MCNC benchmark circuits ("Circuit"), the numbers of modules ("# of Mod."), the numbers of symmetry modules ("# of Sym. Mod."), and the total module areas ("Mod. Area"). Table 2.4 lists the names of the industry benchmark circuits ("Circuit"), the numbers of modules ("# of Mod."), the numbers of symmetry modules ("# of Sym. Mod."), and the total module areas ("Mod. Area").

Based on simulated annealing, a left-skewed HB*-tree was constructed as the initial solution. The initial temperature T_0 was calculated by (2.5), where Δ_{avg} is the average uphill cost and P is the initial probability to accept uphill solutions. During the simulated annealing process, the temperature was reduced at the rate of 0.9 for each subsequent pass, and 20,000 iterations were performed at each temperature/pass.

$$T_0 = -\Delta_{avg} / \ln P. \tag{2.5}$$

In the first set of experiments, the HB*-tree is compared with the following works: sequence pairs [2], segment trees [3], TCG-S [22], and sequence pairs with dummy nodes [31]. Table 2.5 lists the names of the MCNC benchmark circuits ("Circuit"), the total areas ("Area"), and the runtimes ("Time") for the aforementioned works and the HB*-tree with area optimization alone, same as the other works, and with simultaneous area and wirelength optimization. The results of the works [2, 3, 22] are taken from the paper [22], and those of [31] are based on the package provided by the authors. The results show that the HB*-tree achieves average area reductions of 3%, 2%, 1%, and 2% over [2], [3], [22], and [31], respectively. Noted that the improvements should not be considered marginal since the other works have pushed the solution quality close to their limits. The main reason for the area improvement over the other works is that the HB*-tree benefits from both the symmetry-island formulation and the short packing time of the proposed floorplan representations. Based on the symmetry-island formulation, the undesired solutions are pruned, and thus the time is saved to search inferior solutions during simulated

Table 2.5 Comparisons of area utilization and CPU times for sequence pair (SP) (on Sun Sparc Ultra-60 433MHz), segment tree (seg. tree) (on Sun Sparc Ultra-60 433MHz), TCG-S (on Sun Sparc Ultra-60 433MHz), sequence pair with dummy nodes (SP w. dummy) (on Pentium4 3.2GHz), and our HB*-tree (on Pentium4 3.2GHz) with area optimization alone, same as the previous works, and with simultaneous area and wirelength optimization (HB*-tree (area + WL)), based on the MCNC benchmarks

	SP [2]		Seg. tree [3]		TCG-S [22]		SP w. dummy [31]		HB*-tree		HB*-tree (area + WL)		
Circuit	Area (mm^2)	Time (s)	Area (mm^2)	Time (s)	Area (mm^2)	Time (s)	Area (mm^2)	Time (s)	Area (mm^2)	Time (s)	Area (mm^2)	HPWL (mm)	Time (s)
apte	48.12	25	47.52	11	47.52	3	46.92	13	46.92	2	47.90	10.20	3
hp	9.84	138	9.71	62	9.71	50	9.43	13	9.35	2	10.10	30.74	16
ami33	1.24	684	1.23	307	1.21	423	1.24	23	1.23	12	1.29	47.23	39
ami49	37.82	2,038	37.31	983	37.04	1,247	38.32	29	36.85	20	41.32	769.99	96
Comparison	1.03	–	1.02	–	1.01	–	1.02	4.09	1.00	1.00	–	–	–

annealing. With the short packing time, it is possible to search for more solutions within the same time limit. Consequently, the HB*-tree has greater possibility to find better solutions in shorter running time. For the running time, the HB*-tree is approximately 4.09× faster than [31]. Since all the other works ran on different platforms, it is not easy to report the speedups of our algorithm. Nevertheless, it is obvious from the table that the HB*-tree runs much faster than the other works.

In the second set of experiments, the HB*-tree is compared with sequence pairs in [2], segment trees in [5], sequence pairs with linear programming in [14], and sequence pairs with dummy nodes in [31]. Table 2.6 lists the names of the industry benchmark circuits, the total areas and the runtime for sequence-pairs, segment trees, sequence-pairs with linear programming, sequence-pairs with dummy nodes, and HB*-tree. The results show that the HB*-tree achieved average area reductions of 7.1%, 6.6%, 1.6%, and 10.3% over [2], [5], [14], and [31], respectively. In some applications, the orientations of analog device modules may not be allowed to be changed. To make fair comparisons with the other works, the HB*-tree was also performed without module rotation. The results show only 2.4% and 4% area overheads without the rotation, compared to the results of sequence pairs with linear programming [14] and the HB*-tree, respectively. For the running time, the HB*-tree achieves significant speedups over the other works, which is approximately 39.88× and 5.68× faster than those in [14] and [31], respectively. Again, the other works [2, 5] ran on different platforms, and thus the corresponding speedups are not reported, yet it is obvious that the HB*-tree runs much faster than the previous works. It is clear from the two experiments that the HB*-tree achieves the best quality and efficiency than all the other works.

Figure 2.21 shows the resulting placement of ami49 with simultaneous area and wirelength optimization, which contains the symmetry group $S = \{(b_{19}, b_{21}), b^s_{30}, b^s_{48}\}$. Figure 2.22 shows the resulting placements of biasynth_2p4g without module rotation, while Fig. 2.23 shows the resulting placements of biasynth_2p4g with module rotation.

2.7 Advanced Symmetry Constraints

For some analog layout applications, the symmetry constraints could be even more complex than what we have considered. The handling of two kinds of such symmetry constraints is summarized in the following:

2.7.1 Multiple Symmetry-Group Alignment

In some analog layouts, the symmetry axes of different symmetry groups are required to be aligned to share a common symmetry axis. To align multiple symmetry groups with respect to a common vertical (horizontal) symmetry axis, a zero-height

Table 2.6 Comparisons of area utilization and CPU times for sequence pair (SP) (on Sun Blade 100 500 MHz), segment tree (seg. tree) (on Sun Blade 100 500 MHz), SP+LP (Pentium4 3.2 GHz), sequence pair with dummy nodes (SP w. dummy) (on Pentium4 3.2 GHz), and HB*-tree (on Pentium4 3.2 GHz), based on two real industry benchmarks

	SP [2]		Seg. tree [5]		SP+LP [14]		SP w. dummy [31]		HB*-tree		
Circuit	Area (10^3 μm^2)	Time (s)	Area (10^3 μm^2)	Time (s)	Area (10^3 μm^2)	Time (s)	Area (10^3 μm^2)	Time (s)	Area w/o mod. rot. (10^3 μm^2)	Area w/o mod. rot. (10^3 μm^2)	Time (s)
biasynth_2p4g	5.40	780	5.40	246	4.96	206	5.57	134	5.15	4.92	22
lnamixbias_2p4g	50.80	2,824	50.30	726	50.15	3,027	52.21	227	50.28	48.63	43
Comparison	1.071	–	1.066	–	1.016	39.88	1.103	5.68	1.040	1	1

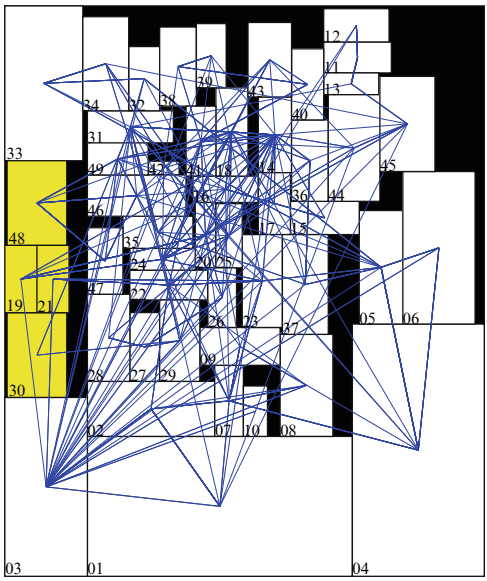


Fig. 2.21 The resulting placement of ami49 with simultaneous area and wirelength optimization, which contains the symmetry group, $S = \{(b_{19}, b_{21}), b_{30}^s, b_{48}^s\}$

(zero-width) dummy block can be inserted right at the left (bottom) of each to-be-aligned symmetry island. A dummy node is then introduced as the parent of the hierarchy node representing the corresponding symmetry island in the HB*-tree, where the hierarchy node is the left (right) child of the dummy node. By adjusting the width (height) of each dummy block, the symmetry islands of different symmetry groups can be aligned with respect to a common vertical (horizontal) symmetry axis. Such an alignment technique is an extension of the work [32].

2.7.2 *Consideration of NonSymmetry-Island Placements*

In addition to the preferred symmetry-island placements in analog layouts, the proposed ASF-B*-trees and HB*-trees can also generate a nonsymmetry-island placement by integrating nonsymmetric modules as a self-symmetric module cluster or a symmetry pair consisting of two module clusters in a symmetry group represented by an ASF-B*-tree. Figure 2.24 shows two examples, including the symmetric placements and the corresponding ASF-B*-trees, which integrate nonsymmetric module clusters into symmetry groups. In Fig. 2.24a, the nonsymmetric modules, b_3 and b_4, form the self-symmetric module cluster C_1 in the symmetry group S_1. After packing the B*-tree representing the placement of the nonsymmetric modules, the representative node $n_{C_1}^r$ is introduced in the ASF-B*-tree representing a symmetric placement of S_1. Similarly, in Fig. 2.24b, the nonsymmetric modules, b_7, b_8, and b_9 form two clusters, C_2 and C_2', as a symmetry pair in the symmetry group S_2. In the corresponding ASF-B*-tree, the representative node $n_{C_2}^r$ is introduced to denote the larger dimensions of the placements of C_2 and C_2'.

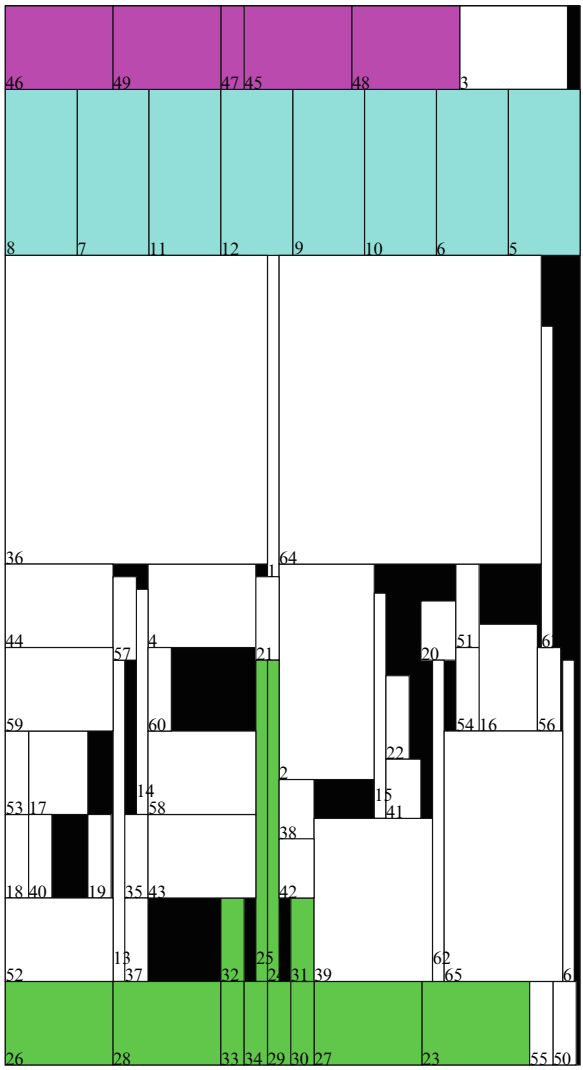


Fig. 2.22 The resulting placement of biasynth_2p4g without module rotation

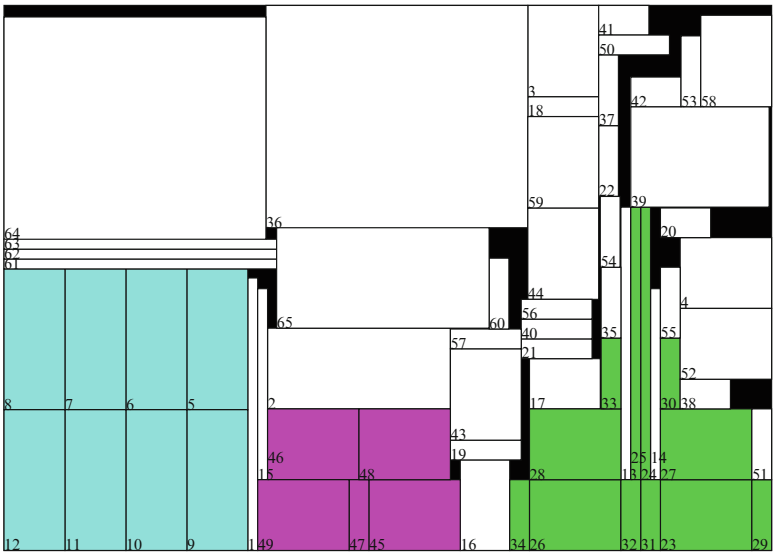


Fig. 2.23 The resulting placement of biasynth_2p4g with module rotation

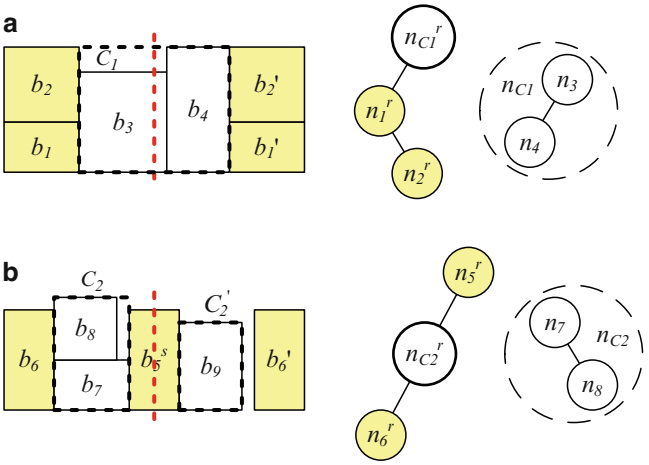


Fig. 2.24 Integrating nonsymmetric modules into symmetry groups. (**a**) The nonsymmetric modules form the self-symmetric module cluster $C_1 = \{b_3, b_4\}$ in the symmetry group $S_1 = \{(b_1, b_1'), (b_2, b_2'), C_1^s\}$. (**b**) The nonsymmetric modules form two clusters, $C_2 = \{b_7, b_8\}$ and $C_2' = \{b_9\}$, as a symmetry pair in the symmetry group $S_2 = \{b_5^s, (b_6, b_6'), (C_2, C_2')\}$

2.8 Hierarchical Constraints

2.8.1 Hierarchical Symmetry

In some fully symmetric analog designs, such as the example in Fig. 2.3, the device layouts should be hierarchically symmetric. A symmetry group S_i may also contain a self-symmetry group S_j^s and/or a symmetry-group pair (S_k, S_k'). Consequently, the top-level symmetry group S_{Top} contains all device modules and other symmetry groups hierarchically. Based on the proposed symmetry-island and tree formulation, a hierarchical tree structure [20] that mixes both the ASF-B*-trees and the HB*-trees can be constructed. The optimized fully symmetric placement with the hierarchical symmetry constraint can then be obtained by searching a desired configuration of the tree structure and packing the trees to form the symmetry islands hierarchically.

2.8.2 Hierarchical Clustering/Proximity

Besides handling the symmetry constraints based on the symmetry-island formulation, the proposed hierarchical framework, HB*-trees, can also effectively manage the hierarchical clustering constraint in analog placement or mixed-signal floorplanning based on the intrinsic hierarchical tree structure.

Let $C = \{C_1, C_2, \ldots, C_l\}$ be a set of device module clusters. Each cluster contains at least two modules, or one module and one of the other clusters, or two of the other clusters. If the cluster C_i contains the cluster C_j, C_i is called a super-cluster, and C_j is called a subcluster. The hierarchical clustering constraint limits all the device modules and/or subclusters of the same super-cluster to a connected placement.

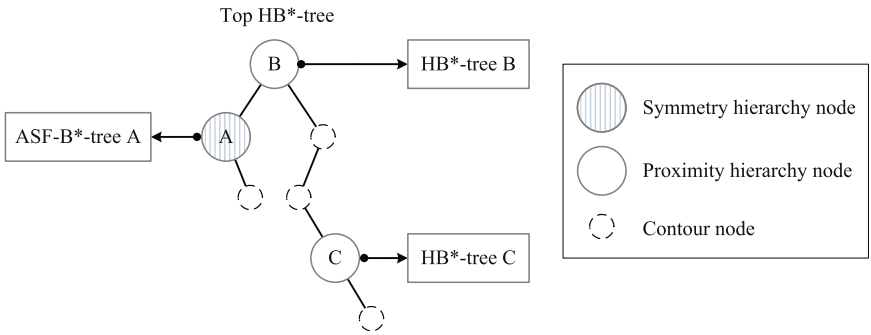


Fig. 2.25 Example HB*-trees modeling the hierarchical floorplan of the design in Fig. 2.2

To formulate the hierarchical clustering constraint using the HB*-trees, each of the hierarchy nodes $n_{C_1}, n_{C_2}, \ldots, n_{C_l}$ denotes a cluster. Each hierarchy node n_{C_i} further contains another HB*-tree to represent the topological relation of the device modules and/or the subclusters in the supercluster denoted by n_{C_i}. After hierarchically constructing the HB*-trees, the placement can be optimized by searching a desired configuration of the HB*-trees while the inner placement of each cluster is connected.

Figure 2.25 shows example HB*-trees modeling the hierarchical placement of the design in Fig. 2.2. Consequently, the number of the HB*-trees will be equal to that of the subcircuits plus one for modeling the top design. When perturbing the HB*-trees, one of the HB*-trees should be selected first, and then any perturbation operation for the B*-tree can be applied to the selected HB*-tree. When converting the HB*-trees to a hierarchical placement, the packing procedure is also similar to that for the B*-tree, which adopts a preorder tree traversal. Once a hierarchy node is traversed, the nodes in the HB*-tree linked by the hierarchy node will be traversed before traversing the next node in the HB*-tree to which the hierarchy node belongs. During the HB*-tree packing, the properties of the proximity constraint should also be considered [20].

The hierarchical framework based on the HB*-tree can easily integrate other placement approaches for different subcircuits with different placement requirements. Besides integrating the ASF-B*-tree, the HB*-tree can also integrate both the corner block list (CBL) and the grid-based approach in [24] for a common-centroid placement, the signal-flow driven approach [17] for the placement of a specific subcircuit with clear signal flows, and other placement approaches.

2.9 Conclusion

This chapter has introduced hierarchical analog placement framework, HB*-tree, with the consideration of layout design hierarchy and hierarchical placement constraints. Different from the existent approaches with at least log-linear-time algorithms, a linear-time packing algorithm has been presented based on the symmetry-island formulation that prunes the solution subspace formed with nonsymmetry-island placements. Experimental results have shown that such approach achieves high quality and runtime efficiency for analog placement.

References

1. F. Balasa, Modeling non-slicing floorplans with binary trees, *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 13–16, 2000
2. F. Balasa and K. Lampaert, Symmetry within the sequence-pair representation in the context of placement for analog design, *IEEE Trans. Computer-Aided Design*, 19(7):721–731, 2000

3. F. Balasa, S. Maruvada, and K. Krishnamoorthy, Efficient solution space exploration based on segment trees in analog placement with symmetry constraints, *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 497–502, Nov 2002
4. F. Balasa, S. Maruvada, and K. Krishnamoorthy, Using red-black interval trees in device-level analog placement with symmetry constraints, *Proceedings of IEEE/ACM Asia South Pacific Design Automation Conference*, pp. 777–782, Jan 2003
5. F. Balasa, S. Maruvada, and K. Krishnamoorthy, On the exploration of the solution space in analog placement with symmetry constraints, *IEEE Trans. Computer-Aided Design*, 23(2): 177–191, 2004
6. Y.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu, B*-trees: a new representation for non-slicing floorplans, *Proceedings of ACM/IEEE Design Automation Conference*, pp. 458–463, 2000
7. E. Charbon, E. Malavasi, and A. Sangiovanni-Vincentelli. Generalized constraint generation for analog circuit design, *Proceedings of ACM/IEEE International Conference on Computer-Aided Design*, 408–414, 1993
8. J. M. Cohn, D. J. Garrod, R. A. Rutenbar, and L. R. Charley. Analog Device-Level Layout Automation. *Kluwer, Dordecht*, 1994
9. J. Cohn, D. Garrod, R. Rutenbar, and L. Carley, Koan/anagram ii: new tools for device-level analog placement and routing, *IEEE J. Solid-State Circuits*, 26(3):330–342, 1991
10. H. Graeb, S. Zizala, J. Eckmueller, and K. Antreich. The sizing rules method for analog integrated circuit design, *Proceedings of ACM/IEEE International Conference on Computer-Aided Design*, 343–349, 2001
11. A. Hastings. The Art of Analog Layout. 2nd Ed. *Pearson Prentice Hall*, 2006
12. D. Jepsen and C. Gelatt Jr., Macro placement by monte carlo annealing, *Proceedings of IEEE International Conference on Computer Design*, pp. 495–498, Nov 1983
13. S. Kirkpatrick, J. Gelatt, C. D., and M. P. Vecchi, Optimization by Simulated Annealing, *Science*, 220(4598):671–680, 1983
14. S. Koda, C. Kodama, and K. Fujiyoshi, Linear programming-based cell placement with symmetry constraints for analog ic layout, *IEEE Trans. Computer-Aided Design*, 26(4):659–668, 2007
15. K. Krishnamoorthy, S. Maruvada, and F. Balasa, Topological placement with multiple symmetry groups of devices for analog layout design, *Proceedings of IEEE International Symposium on Circuits and Systems*, pp. 2032–2035, May 2007
16. K. Lampaert, G. Gielen, and W. Sansen, A performance-driven placement tool for analog integrated circuits, *IEEE J. Solid-State Circuits*, 30(7):773–780, 1995
17. D. Long, X. Hong, and S. Dong. Signal-path driven partition and placement for analog circuit, *Proceedings of ACM/IEEE Asia South Pacific Design Automation Conference*, 694–699, 2006
18. J.-M. Lin and Y.-W. Chang, TCG: A transitive closure graph based representation for general floorplans, *IEEE Trans. VLSI Systems*, 13(2):288–292, 2005
19. P.-H. Lin and S.-C. Lin, Analog placement based on novel symmetry-island formulation, *Proceedings of ACM/IEEE Design Automation Conference*, pp. 465–470, Jun 2007
20. P.-H. Lin and S.-C. Lin, Analog placement based on hierarchical module clustering, *Proceedings of ACM/IEEE Design Automation Conference*, pp. 50–55, Jun 2008
21. J.-M. Lin, H.-E. Yi, and Y.-W. Chang, Module placement with boundary constraints using B*-trees, *IEE Proceedings – Circuits, Devices and Systems*, 149(4),:251–256, 2002
22. J.-M. Lin, G.-M. Wu, Y.-W. Chang, and J.-H. Chuang, Placement with symmetry constraints for analog layout design using TCG-S, *Proceedings of IEEE/ACM Asia South Pacific Design Automation Conference*, vol. 2, pp. 1135–1138, Jan 2005
23. P.-H. Lin, Y.-W. Chang, and S.-C. Lin, Analog placement based on symmetry-island formulation, *IEEE Trans. Computer-Aided Design*, 28(6):791–804, 2009
24. Q. Ma, E. F. Y. Young, K. P. Pun. Analog placement with common centroid constraints, *Proceedings of ACM/IEEE International Conference on Computer-Aided Design*, 579–585, 2007
25. E. Malavasi, E. Charbon, E. Felt, and A. Sangiovanni-Vincentelli, Automation of ic layout with analog constraints, *IEEE Trans. Computer-Aided Design*, 15(8):923–942, 1996

26. S. Maruvada, A. Berkman, K. Krishnamoorthy, and F. Balasa, Deterministic skip lists in analog topological placement, *Proceedings of IEEE International Conference on ASIC*, vol. 2, pp. 834–837, 2005
27. T. Massier, H. Graeb, and U. Schlichtmann. Sizing rules for bipolar analog circuit design, *Proceedings of ACM/IEEE International Conference on Design, Automation and Test in Europe*, 140–145, 2008
28. H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. VLSI module placement based on rectangle-packing by the sequence-pair, *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 15(12):1518–1524, 1996
29. M. Pelgrom, A. Duinmaijer, and A. Welbers, Matching properties of mos transistors, *IEEE J. Solid-State Circuits*, 24(5):1433–1439, 1989
30. M. Strasser, M. Eick, H. Graeb, U. Schlichtmann, and F. M. Johannes. Deterministic analog circuit placement using hierarchically bounded enumeration and enhanced shape functions, *Proceedings of ACM/IEEE International Conference on Computer-Aided Design*, 2008
31. Y.-C. Tam, E. F. Y. Young, and C. Chu, Analog placement with symmetry and other placement constraints, *Proceedings of IEEE/ACM International Conference on Computer-Aided Design*, pp. 349–354, Nov 2006
32. M.-C. Wu and Y.-W. Chang, Placement with alignment and performance constraints using the B*-tree representation, *Proceedings of IEEE International Conference on Computer Design*, pp. 568–571, Oct 2004
33. G.-M. Wu, Y.-C. Chang, and Y.-W. Chang, Rectilinear block placement using B*-trees, *ACM Transactions on Design Automation of Electronics Systems*, 8(2):188–202, 2003
34. L. Zhang, C.-J. R. Shi, and Y. Jiang, Symmetry-aware placement with transitive closure graphs for analog layout design, *Proceedings of IEEE/ACM Asia South Pacific Design Automation Conference*, pp. 180–185, Mar 2008

Chapter 3
Deterministic Analog Placement by Enhanced Shape Functions

Martin Strasser, Michael Eick, Helmut Graeb, and Ulf Schlichtmann

Abstract For analog integrated circuits, generating a layout represents the bottleneck in the design flow. To automate the layout step, it is necessary to create placements with respect to various constraints automatically. Since the constraints can be numerous, an automatic generation of the layout constraints is crucial as well. In this chapter, a comprehensive and deterministic methodology for analog layout design automation is presented. An approach to automatically generate constraints for analog circuits is described. It recognizes building blocks, e.g., current mirrors, and symmetry conditions in the circuit and, with prioritized rules, generates constraints and hierarchy information. Then, a placement algorithm, called "Plantage", is presented, which is capable to handle all relevant constraints. It uses the hierarchy information of the previous step to guide an enumeration process. Plantage calculates a Pareto front of placements with respect to different aspect ratios. The results show high quality in terms of area and postlayout circuit performance.

3.1 Introduction

Modern integrated circuits often contain digital as well as analog parts. The design of the analog part is usually a time consuming step. While digital circuits can be designed using a variety of layout approaches, analog circuits are still manual and error-prone tasks for the designers in many cases.

The placement and the routing of an analog circuit have a severe impact on the function and performance. Thus, layout constraints are defined to make sure that the circuit fulfills the performance specifications. As an example, unbalanced parasitics, being a result of asymmetrical layout, may be detrimental to the power supply rejection ratio or the offset voltage of an analog amplifier.

The number and diversity of constraints imposed on an analog circuit prevent approaches used in the digital domain from being used for analog design.

M. Strasser (✉)
Institute for Electronic Design Automation, Technische Universität München, Munich, Germany
e-mail: strasser@tum.de

H.E. Graeb (ed.), *Analog Layout Synthesis: A Survey of Topological Approaches*,
DOI 10.1007/978-1-4419-6932-3_3, © Springer Science+Business Media, LLC 2011

3.1.1 Definitions

In this section, the terms used throughout this chapter are defined.

Definition 3.1 (Device). A device d is an elementary part of the circuit, e.g., a transistor. The set of all devices is referred to as $\mathcal{D}$ in this chapter.

Definition 3.2 (Module). A module m is the smallest item the placer has to deal with. It is represented by a rectangle on the placement plane. Modules will be referred to by Latin lowercase letters in the following.

A device consists of one to several modules. For example, several physical transistors (represented by different modules) may be interconnected to form one logical transistor (represented by a single device). $\mathcal{M}$ is the set of modules. For every module $m \in \mathcal{M}$, the lower left corner coordinates are described by x_m and y_m, and the width and height are referred to as w_m and h_m, respectively.

Definition 3.3 (Center of gravity of a module). The coordinates of the center of gravity (COG) of a module are defined as

$$x_{\mathrm{COG}}(m) = x_m + \frac{w_m}{2}, \tag{3.1}$$

as well as

$$y_{\mathrm{COG}}(m) = y_m + \frac{h_m}{2}. \tag{3.2}$$

Using vectors, the center of gravity can be formulated as $\mathbf{x}_{\mathrm{COG}}(m)$:

$$\mathbf{x}_{\mathrm{COG}}(m) = \begin{pmatrix} x_{\mathrm{COG}}(m) \\ y_{\mathrm{COG}}(m) \end{pmatrix}. \tag{3.3}$$

Definition 3.4 (Distance between modules). The distance of module m from module n is denoted by $d(m,n)$, defined as the minimum of the vertical and the horizontal distance between m and n:

$$d(m,n) = \min(d_{\mathrm{hor}}(m,n), d_{\mathrm{vert}}(m,n)), \tag{3.4}$$

$$d_{\mathrm{hor}}(m,n) = \max\left(|x_{\mathrm{COG}}(m) - x_{\mathrm{COG}}(n)| - \frac{w_m + w_n}{2}, 0\right), \tag{3.5}$$

$$d_{\mathrm{vert}}(m,n) = \max\left(|y_{\mathrm{COG}}(m) - y_{\mathrm{COG}}(n)| - \frac{h_m + h_n}{2}, 0\right). \tag{3.6}$$

Definition 3.5 (Group). A group G is defined as a set of modules or as a set of groups, which are intended to be placed in close proximity.

A group G^0 consisting only of modules, which means $G^0 \subseteq \mathcal{M}$, is called a *basic group*. The set of all basic groups is called $\mathcal{G}^0$, which is the power set of $\mathcal{M}$ without the empty set:

$$G^0 \in \mathcal{G}^0 = \mathbf{P}(\mathcal{M}) \backslash \{\varnothing\}. \tag{3.7}$$

A group G^i, $i \geq 1$, only contains other groups G_j^{i-1}, G_k^{i-1}. It is called a *hierarchical group of hierarchy level i*. The set of all possible groups for hierarchy level i is called $\mathcal{G}^i$, which is the power set of $\mathcal{G}^{i-1}$ without the empty set:

$$G^i \in \mathcal{G}^i = \mathbf{P}(\mathcal{G}^{i-1}) \backslash \{\varnothing\}. \tag{3.8}$$

A function modulesOf$(G) \subseteq \mathcal{M}$ is defined, which returns the set of all modules of group G. It is defined recursively:

$$\text{modulesOf}(G) = \begin{cases} G, & \text{if } G \in \mathcal{G}^0 \\ \bigcup\limits_{C \in G} \text{modulesOf}(C) & \text{else.} \end{cases} \tag{3.9}$$

Each module is allowed to be part of only one group:

$$\forall_{j \neq k} \text{modulesOf}(G_j^i) \cap \text{modulesOf}(G_k^i) = \varnothing. \tag{3.10}$$

The set of all possible groups $\mathcal{G}$ can be defined as:

$$\mathcal{G} = \bigcup_i \mathcal{G}^i. \tag{3.11}$$

All groups are referred to by Latin capital letters in this chapter.

Definition 3.6 (Center of gravity of groups). Similar to Definition 3.3 of the center of gravity of modules, a center of gravity of a group can be defined:

$$\mathbf{x_{COG}}(G) = \frac{\sum_{m \in \text{modulesOf}(G)} w_m \cdot h_m \cdot \mathbf{x_{COG}}(m)}{\sum_{m \in \text{modulesOf}(G)} w_m \cdot h_m}. \tag{3.12}$$

Definition 3.7 (Module variants). For any module $m \in \mathcal{M}$, there may be several different alternative layouts. The bounding rectangle of an alternative layout of a module is called a *module variant*. The set of all module variants for m is referred to as $\mathcal{V}_m$, a single module variant is denoted by v_m.

Examples for module variants are different numbers of gate fingers of transistors.

3.1.2 Analog Circuit Placement Requirements

In common layout approaches, the placement is generated before routing the circuit. The placement is subject to various constraints. These constraints are formulated to improve the matching of devices, which are intended to be identical by design. *Matching* can be considered to be an umbrella term of different means to reduce the influence of variations of the process and operating conditions, as well as parasitics.

- *Variant constraints* restrict the combination of possible realizations (variants) of circuit modules to ensure matching of devices. When a module has a set of possible variants, the placement algorithm faces a higher degree of freedom and better placements can be achieved. In practice, however, the combination of the different variants is not completely free. For example, a variant constraint is formulated to make sure that both transistors of a differential pair are realized with the same number of gate fingers.
- *Device-proximity constraints* are used to make sure that a group of modules is placed in close proximity. Due to local variations during the fabrication process, the parameters of the devices show unwanted deviations from each other (also referred to as "mismatch"), which can result in performance degradation. Variations in the operating conditions, such as supply voltage or temperature, may have the same effect. Placing matched devices in close proximity can limit the impact of these variations [1].
- *Symmetry constraints* are used for geometric and electrical reasons. They allow for symmetric routing and reduce the sensitivity to on-die thermal gradients. In addition, parasitic resistors and capacitors can be balanced on both halves of a differential circuit [2, 3].
  The module i' denotes a module that is to be placed symmetrically to some module i. For self-symmetrical modules, i is equal to i'. Figure 3.1a shows an example of symmetry constraints, where all modules are arranged with respect to a vertical symmetry axis. For a vertical symmetry, linear equations can be formulated as follows:

$$\forall_i \left(\frac{1}{2} \left(x_i + \frac{w_i}{2} + x_{i'} + \frac{w_{i'}}{2} \right) = x_{\text{sym}} \right), \tag{3.13}$$

$$\forall_i \left(y_i + \frac{h_i}{2} = y_{i'} + \frac{h_{i'}}{2} \right). \tag{3.14}$$

  The equations for a horizontal symmetry axis can be defined analogously.
- *Common centroid constraints* are formulated to arrange the centers of gravity for groups of modules. It is a widely used constraint, which improves the beneficial

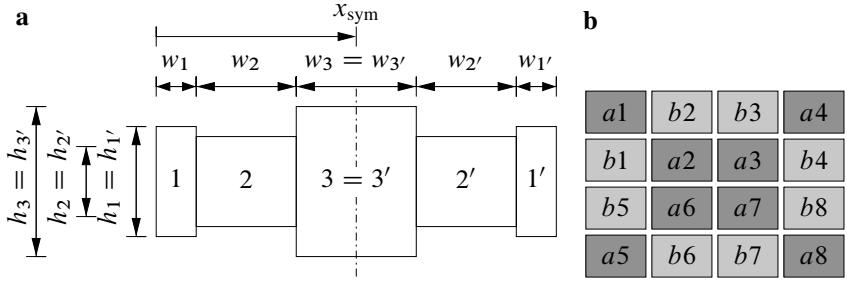


Fig. 3.1 Placement with symmetry (**a**) and common centroid (**b**) constraint

effects of the symmetry constraint [2]. For example, a differential pair can be formed by 16 transistors and arranged as shown in Fig. 3.1b. The transistors a1-a8 (b1-b8) are connected in parallel. All transistors have the same size and the two groups of transistors share the same center of gravity. For these two groups of modules, A and B, a common centroid constraint can be defined as follows:

$$\mathbf{x}_{\mathbf{COG}}(A) = \mathbf{x}_{\mathbf{COG}}(B). \tag{3.15}$$

Beyond those constraints, it is necessary to consider additional constraints for minimum distances for technological reasons:

- *Minimum distance constraints* are formulated if modules must not abut on each other directly, but need a minimum distance. These constraints can be defined for technological reasons. There are two different types of minimum distance constraints, linear minimum distance constraints, and piecewise-linear minimum distance constraints. As an example, some transistors within the same well may abut directly, but a minimum distance is required from transistors outside of the well. Formally, this constraint can be defined as a linear inequality. Piecewise-linear minimum distance constraints are required for special devices.
  - *Linear minimum distance constraints:* To manufacture a CMOS circuit, p-channel transistors usually need to be located in wells. Furthermore, a set of modules can be surrounded by a guard ring to prohibit latch-up effects. For both wells and guard rings, area needs to be reserved in the direct surrounding. Figure 3.2 depicts a set of modules A with a guard ring. For the example of Fig. 3.2, the guard ring has a width of d_{g}. Thus, a minimum distance constraint is formulated to make sure that all modules m within the set of modules A keep a distance $d(m,n) \geq d_{\min}(m,n) = d_{\mathrm{g}}$ from all other modules ($n \notin A$):

$$\bigvee_{m \in A, n \notin A} d(m,n) \geq d_{\min}(m,n), \quad d_{\min}(m,n) \geq 0. \tag{3.16}$$

    The constant minimum distance between m and n is denoted by $d_{\min}(m,n)$. The maximum distance required from module m to any other module is denoted by $d_{\max}(m)$:

$$\bigvee_{m \neq n} d_{\max}(m) \geq d_{\min}(m,n). \tag{3.17}$$

    Since this minimum distance constraint is defined by a linear inequality, we refer to this constraint as a *linear minimum distance constraint*.

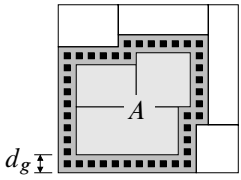


Fig. 3.2 Group A surrounded by a guard ring

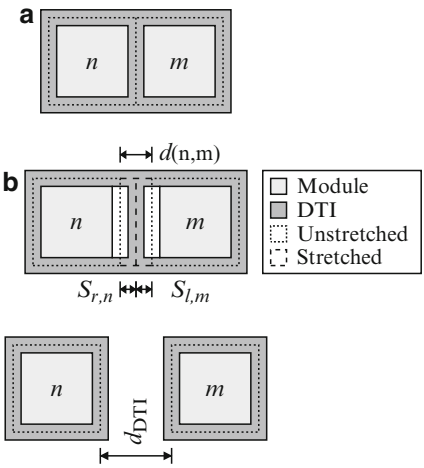


Fig. 3.3 DTI transistors at different distances: (**a**) directly abutting, (**b**) with stretched DTIs, (**c**) with extended minimum distance

– *Piecewise-linear minimum distance constraints* are formulated for special devices which require more complex minimum distance constraints. A good example for such constraints are transistors with deep trench isolation (DTI). DTI transistors can be used for switching high voltages on a chip [4]. Each such transistor is directly surrounded by a DTI. Since the manufacturing process of DTIs is complex, different layout restrictions apply. Figure 3.3 illustrates restrictions for distances between DTI transistors, which must be considered during the placement step.
Neighboring transistors can share DTIs if they are placed in close proximity. Figure 3.3a shows two modules being placed side by side, with a distance $d(n, m) = 0$. The vertical DTI in the middle is shared by both modules.
Since the distance from the DTI to the transistor influences its electrical parameters, it is not allowed to arbitrarily modify the position, sizing, or shape of the DTIs. In common processes, the DTIs surrounding transistors have to be rectangular. It is allowable to stretch their width and height by a few percent. In case the distance between the transistors needs to be increased (for example, due to a symmetry constraint), the transistors can still share a DTI between them within certain boundaries. Figure 3.3b shows two transistors sharing the vertical DTI in the middle, with both DTIs stretched to the limit on the right ($s_{r,n}$) and left ($s_{l,m}$) side of n and m, respectively. In this case, the total stretching of both DTIs is $d(n, m) = s_{r,n} + s_{l,m} = s_{\max}$.
In case the required distance between the two transistors exceeds the stretching limit of $s_{\max}$, the DTI between them can no longer be shared. Due to manufacturing considerations, there is a minimum distance defined between two parallel trenches. For this reason, the transistors must keep an extended minimum distance of d_{DTI} in that case, as shown in Fig. 3.3c.
Summing up, there are three ranges of the distance between two DTI transistors. The first range starts from directly abutting transistors, with a distance

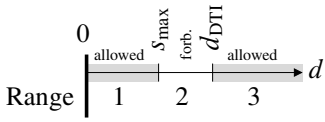


Fig. 3.4 Allowable distance ranges between two DTI transistors

of 0, sharing a DTI. It ends when the DTIs of both transistors are stretched to the limit $s_{\max}$. There, the second range starts and ends at the minimum distance between two DTIs, i.e., d_{DTI}. The second range can be considered as a forbidden zone for the distance between two DTI transistors. The third range is defined from the positive end of the second range to infinity. The ranges of a distance d are depicted in Fig. 3.4, and defined by (3.18):

$$\underbrace{d \leq s_{\max}}_{\text{Range 1}} \quad \vee \quad \underbrace{d \geq d_{\mathrm{DTI}}}_{\text{Range 3}}. \tag{3.18}$$

These minimum distance constraints can no longer be formulated by single inequalities. There are at least two inequalities for disjoint ranges. Thus, we refer to these constraints as *piecewise-linear minimum distance constraints*.

The placement constraints are also crucial for the routability of the design. For example, in differential circuits, the devices often need to be connected by symmetric wires to balance parasitic resistances and capacitances. A prerequisite for symmetric routing is a symmetric placement.

In general, the layout of a circuit needs to be compact for economic reasons. As a conclusion, successful analog layout automation algorithms must produce compact results, considering various constraints.

3.1.3 Context of This Work

A number of methods for automatic placement constraint generation have been published.

The methods proposed in [5–7] evaluate sensitivity analyses to identify parasitics, matching and symmetry constraints of a circuit. The authors of [8] proposed a method that classifies the nets of a circuit according to their susceptibility. This classification is then used to identify analog building blocks (e.g., current mirrors) and their matching constraints. The algorithms presented in [9–12] use a structural analysis of the circuit to find symmetry in the circuit. The underlying subgraph isomorphism problem is solved by graph labeling [9, 12] and recursive detection of symmetric pairs [10, 11].

The generation of sizing constraints is a related problem, because both have to consider matching requirements. The authors of [13] present a method to find basic building blocks defined by a library from the netlist of a circuit. The building blocks are used to assign sizing constraints.

Since the beginning of the 1980s, different approaches for integrated analog circuit placement have been published. The approaches can be classified by the representation they use to store the location of the modules.

The approaches of [14–17] use absolute coordinates of the modules. All constraints are directly formulated using the coordinates. These approaches generate placements using simulated annealing [18]. Minimum distance constraints can be considered by a special term in the cost function, being 0 if the constraints are met, and greater than 0, if the constraints are violated. Because of the high dimensionality of the search space of $\mathbb{R}^{2N}$ for N modules, the computation times based on this representation are high.

In contrast, topological representations have a much smaller search space [19], while still being able to store all *admissible* [20] placements. These representations do not allow overlaps. Prominent topological representations include the O-Tree [20], B*-tree [21, 22], H/ASF-B*-Tree [23] (see Chap. 2), sequence pair [24], bounded sliceline grid [25], corner block list [26, 27], and TCG-S [28]. The placer Plantage, presented in this chapter, is based on B*-trees.

An example of a B*-tree and the corresponding placement is shown in Fig. 3.5. Each node in a B*-tree represents a module. B*-trees use topological relations to encode a placement [21]. The rules to generate a placement without constraints can be summarized as follows: Any node in a B*-tree may have up to two child nodes: One left and one right node. Each node represents a module. The module of a left child is placed above the module of the parent node. The module of the right child is placed right of the module of the parent node. In case the y projections of two modules overlap, the module which comes first in a preorder traversal of the B*-tree is placed left of the other module. The preorder traversal of the tree in the example of Fig. 3.5 is ABCD. A and B are placed to the left of C, B is placed to the left of D. The resulting placement is compacted to the lower left corner.

As described in Chap. 1 of this book, constraints can be used efficiently to restrict the solution space [22, 29–32]. The authors of these papers propose Simulated Annealing algorithms that consider symmetry constraints with a restricted solution space using O-trees [29], B*-trees [22], sequence pairs [30, 31], and sequence pairs with Johnson's priority queue [32]. In [23], a different approach is presented, based on two modifications of the B*-tree: the first is to handle symmetry constraints with so-called symmetry islands and the second to combine these symmetry islands with the rest of the modules. The authors of [33] presented a placement algorithm with symmetry and other placement constraints. The concept of dummy nodes in constraint graphs is introduced to fulfill symmetry constraints. All previous works mentioned here use simulated annealing to optimize placement. In contrast, Plantage is deterministic.

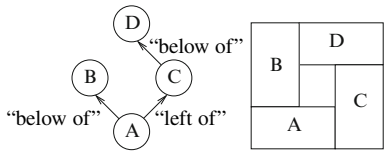


Fig. 3.5 B*-Tree and its corresponding placement

3.1.4 Contributions

Analog integrated circuits are hierarchically built [1, 34]. For example, an operational amplifier can be decomposed into a differential input stage and a output stage. The input stage in turn can be divided into a differential pair and several current mirrors. This hierarchy can be described as a hierarchy tree, whose leaf nodes are the devices of the circuit. The root node of this tree represents the whole circuit. All inner nodes of this hierarchy tree including the root node are denoted as groups.

In this chapter, we first introduce a new method that automatically determines the hierarchy tree of an analog circuit. This process is controlled by the required matching, symmetry, and proximity constraints. These constraints are determined automatically and are modeled in a placement requirement graph. Furthermore, we show how the hierarchy tree can be efficiently used together with the constraints to formulate so-called hierarchical placement rules.

Since B*-trees can generate all admissible placements, a complete enumeration would yield the optimal solution. However, this is not practicable due to complexity problems, because the number of B*-trees increases more than exponentially with the number of modules. Plantage uses the hierarchy to bound the enumeration. For small subparts of the circuit, all possibilities are enumerated. These partial solutions are then combined, guided by the hierarchy, to generate placements for the complete circuit. If the partial solutions were joined together using their bounding boxes, the area usage would deteriorate. Thus, a new concept is used, which calculates a new placement as a sum of the B*-trees of the partial placements. This concept is designed to avoid white space while the two B*-trees are assembled to one. The result of Plantage is a set of area-optimal placements (shapes) with different aspect ratios.

The placement algorithm starts with basic groups. For the basic groups, the complete solution space is enumerated. Since B*-trees are used for enumeration, the process can be accelerated using feasibility checks, as described in Chap. 1. An algorithm is proposed in Sect. 3.3 to generate a placement for a given B*-tree considering all constraints for analog circuits. To store all Pareto optimal placements for the basic groups, *enhanced shape functions* are used and described in Sect. 3.4.

After all possible placements for the basic groups have been calculated, the algorithm steps up to next level in the hierarchy. The results of the previously calculated placements for the basic groups are then combined. For the current hierarchy level, the Pareto optimal results are stored in an enhanced shape function. Suboptimal combinations are removed in every hierarchy level to limit the computational effort in subsequent steps. This methodology is repeated until the highest hierarchy level is reached, covering the whole circuit. Finally, the enhanced shape function of the whole circuit represents the Pareto front of optimal layouts with different aspect ratios, in contrast to other state-of-the-art approaches, producing a single layout. This enables the designer to choose among different valid designs having different aspect ratios.

The approach presented in this chapter has the following key features:

- Generation of placement rules based on automatically detected symmetry conditions and basic building blocks
- First approach, known to the authors, introducing a hierarchy concept for placement rules of analog integrated circuits, which provides:
  - A prioritization of placement rules by importance,
  - A hierarchical clustering of the circuit,
  - Cluster-specific constraint information.
- Automatic placement considering all beforehand generated constraints.
- Computation of a set of possible placements with different aspect ratios instead of a single solution.
- Based upon a nonslicing topological placement structure, the B*-tree.
- Full enumeration of basic groups, guided by the hierarchy of the circuit.
- Deterministic algorithm, suitable for parallelization.
- Variant selection is integrated seamlessly in the enumeration.

This chapter is organized as follows: Section 3.2 describes an approach to automatically generate the hierarchy tree as well as constraints, based on a detection of basic building blocks in the circuit. Section 3.3 describes the algorithm to generate a placement for a given B*-tree. In Sect. 3.4, enhanced shape functions are defined. Section 3.5 describes the comprehensive hierarchical placement approach. Section 3.6 shows experimental results. A conclusion is given in Sect. 3.7.

3.2 Placement Constraint Generation

Our constraint generation method is based on a set of five different placement requirements concerning symmetry, matching, and proximity. At the beginning of this section, this set and an associated order, describing the importance of each requirement, are introduced. After that, the generation method itself is described. Figure 3.6 gives an overview. It uses a set of recognized building blocks, like, current mirrors, and symmetry conditions to generate a graph of placement requirements with respect to symmetry, matching, and proximity (SMP graph). Based on the SMP graph, a tree of hierarchical symmetry, matching, and proximity groups is generated (HSMPG tree).

3.2.1 Placement Requirements

3.2.1.1 Types of Placement Requirements

In the following, five different types of placement requirements are distinguished.

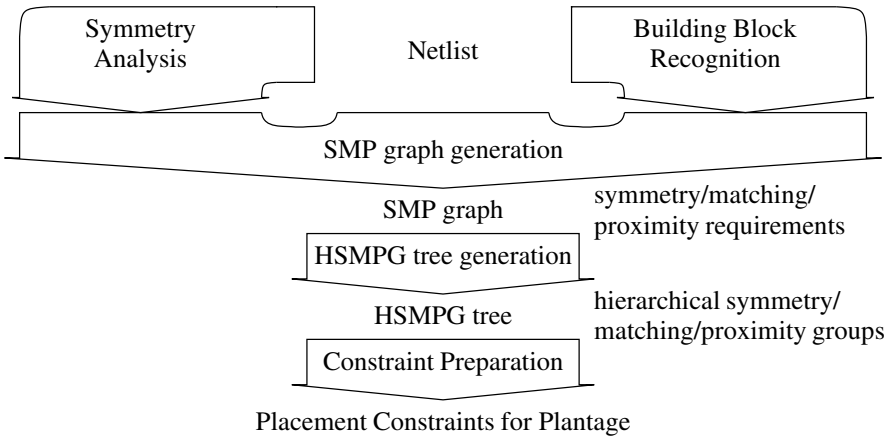


Fig. 3.6 Overview of the placement constraint generation method

Table 3.1 Symbols and definitions of placement requirement types

Symbol	Definition
M_S	Device matching requirement between devices of a symmetric device pair
M_B	Device matching requirement between devices of a building block
P_B	Proximity requirement of a building block
S	Symmetry requirement
P_N	Proximity requirement from the netlist

Definition 3.8 (Placement requirement type). The type t of a placement requirement must be an element of

$$T = \{M_B, M_S, S, P_B, P_N\}.$$

Table 3.1 defines the different types $t \in T$.

Types M_B and M_S describe matching requirements between devices of a circuit. All devices subject to the same matching requirement must have equal electrical properties. Matching requirements can originate from either building blocks (M_B), e.g., current mirrors, or concern devices that are symmetric to each other (M_S). Symmetry requirements (type S) represent conditions to the device coordinates, i.e., (3.13) and (3.14). Types P_B and P_N define a requirement for close spatial proximity. These requirements can originate either from building blocks (P_B) or from the connections in the netlist (P_N).

3.2.1.2 Importance Order

Different placement requirements exist for every device in the circuit. These multiple requirements may impose conflicts on the placement. In the following, a priority

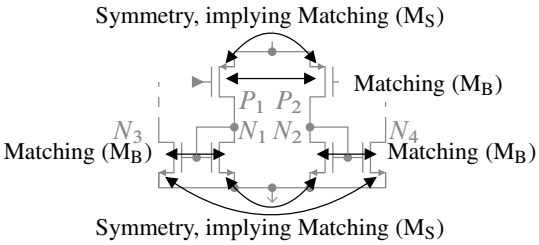


Fig. 3.7 Detail of example circuit (Fig. 3.10) with constraint requirements

order for the five types of matching, proximity, and symmetry requirements is formulated. This priority order is used by our HSMPG tree generation algorithm to avoid conflicts.

In general, matching requirements, which are created from symmetric device pairs (M_S) and building blocks (M_B), are most important. Symmetric device pairs represent the matching between the two parts of a differential circuit, which is more important than the matching inside each part.

This is illustrated by the following example. Figure 3.7 shows a detail of the circuit from Fig. 3.10. Transistors P_1 and P_2 form a differential pair and are therefore subject to a matching requirement. N_1 and N_3, as well as N_2 and N_4, respectively, form current mirrors, which require matching of their respective devices. The part of the circuit shown in Fig. 3.7 is fully differential, which demands symmetry between P_1 and P_2, between N_1 and N_2, and between N_3 and N_4. This implies a pairwise matching of these devices in addition, resulting in four matching requirements between the transistors N_1 to N_4. Not all of these matching constraints are equally important, as illustrated in the following: In case the matching from symmetry between N_1 and N_2, as well as between N_3 and N_4 is disregarded, performances such as offset error are degraded. These performances are considered as critical for most applications. If these transistors are matched, then a mismatch between N_1 and N_3 equals a mismatch between N_2 and N_4. This means the mismatch inside the building blocks is equal. Consequently, the operating points in both parts of the differential circuit are affected equally, leading to a degradation of, e.g., the gain, which is considered as less critical for most applications. Overall, the matching requirements among the internal transistors of the current mirrors (N_1,N_3) and (N_2,N_4), are less critical than the matching requirements emerging from symmetry between N_1 and N_2, as well as between N_3 and N_4.

Symmetry requirements always affect whole building blocks. Therefore, they are not harmed by proximity requirements of type P_B, which exist only inside building blocks. Therefore, a higher priority is assigned to proximity requirements P_B than to symmetry requirements S. The remaining proximity requirements of the netlist P_N are least important.

Definition 3.9 (Importance order and importance ordered type set T_I). The importance order $\prec$ is a strict order of the set of constraint types T. It holds:

$$M_S \prec M_B \prec P_B \prec S \prec P_N. \tag{3.19}$$

The corresponding importance ordered type set T_I is defined as:

$$T_I := (T, \prec). \tag{3.20}$$

3.2.2 SMP Graph and Its Generation

Definition 3.10 (SMP graph). The SMP graph $\mathcal{G}_{\text{SMP}}(\mathcal{N}_{\text{SMP}}, \mathcal{E}_{\text{SMP}}, t_{\text{SMP}})$ is an undirected graph of placement requirements with respect to symmetry, matching, and proximity. The nodes $\mathcal{N}_{\text{SMP}}$ are formed by the devices of the circuit. Two nodes $e.a \in \mathcal{N}_{\text{SMP}}$ and $e.b \in \mathcal{N}_{\text{SMP}}$ with $e.a \neq e.b$ are connected by an edge $e \in \mathcal{E}_{\text{SMP}}$ iff they are subject to the same requirement. The function $t_{\text{SMP}} : \mathcal{E}_{\text{SMP}} \rightarrow T$ defines the type of requirement of each edge. An SMP graph is a multigraph [35] allowing multiple edges between two nodes.

The SMP graph is initialized with proximity requirements from the netlist.

For the routing of analog circuits, it is beneficial that nets are as short as possible. To obtain this, a proximity requirement is defined between each pair of devices connected to the same net n, leading to a complete subgraph n (Fig. 3.8).

Figure 3.9 shows the initial SMP graph for the example circuit from Fig. 3.10. The SMP graph is then successively filled with further matching, symmetry, and proximity requirements. These requirements are determined through building block recognition and symmetry analysis.

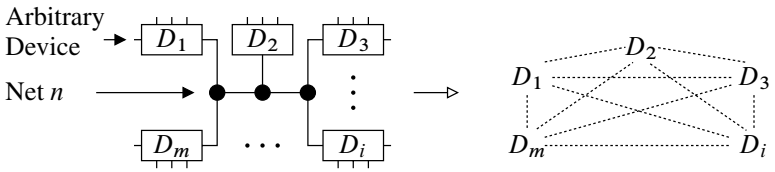


Fig. 3.8 Edges in the SMP graph representing a proximity requirement from one net of the netlist

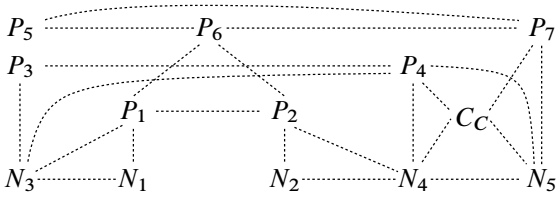


Fig. 3.9 Initial SMP graph for the circuit from Fig. 3.10. It contains only proximity requirements originating from the netlist

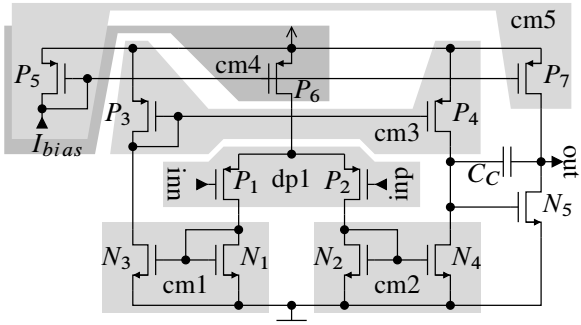


Fig. 3.10 A simple amplifier with recognized building blocks

3.2.2.1 Building Block Recognition

In the second step, the inherent building blocks of a circuit are identified.

The building block recognition is based on the algorithm published by the authors of [13]. It recognizes the building blocks from a given library by finding subgraph isomorphism.

For each element of the library, that has an independent function, placement requirements for matching and proximity are defined in addition. Figure 3.11 shows the corresponding assignments of the library elements.

For the two transistor building blocks, differential pair, level-shifter and simple current mirror a matching of their two transistors is required. The complex current mirrors, Cascode current mirror, four transistor current mirror, and wide-swing current mirror, require a matching of the lower transistors T_1, T_2 and of the upper transistors T_3,T_4. The routing requires the elements of the building block to be in close spatial proximity in the final layout. Therefore, a building block proximity requirement is defined between T_1,T_3 and T_2,T_4, respectively.

For the example from Fig. 3.10, the algorithm recognizes five simple current mirrors `cm1` to `cm5` and one differential pair `dp1`. Figure 3.12 shows the additional edges, generated in the SMP graph for building blocks, which represent matching and proximity placement requirements.

3.2.2.2 Symmetry Analysis

The symmetry analysis step determines symmetry conditions within the devices of a circuit. All symmetric device pairs having the same symmetry axis form a symmetry compound. The set of all symmetry compounds of a circuit is denoted by S. The analysis algorithm is similar to the one presented in [10].

Two types of requirements are generated for each symmetry compound C (Fig. 3.13). A matching requirement of type M_S is generated for each symmetric device pair of a symmetry compound. A symmetry requirement of type S is generated

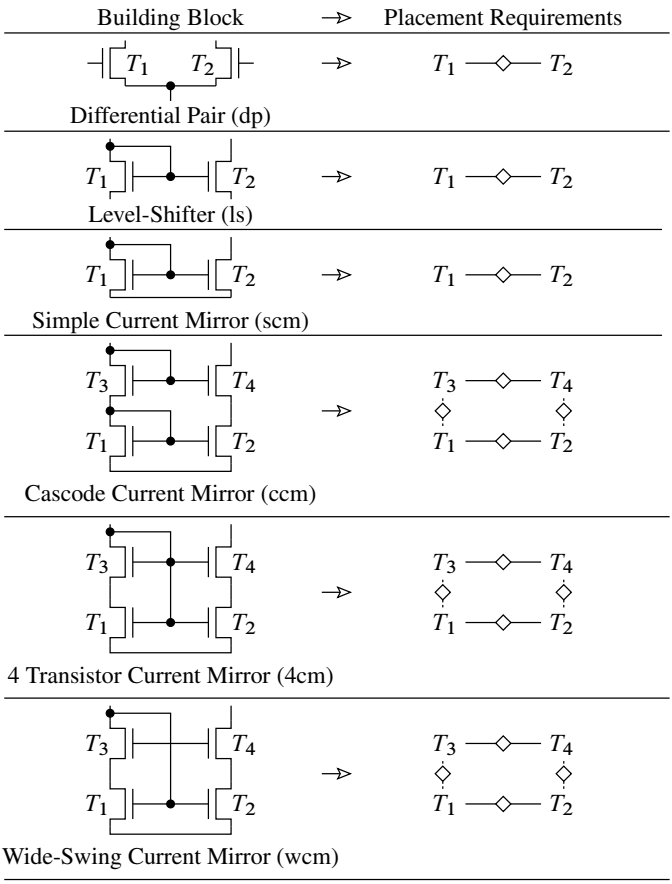


Fig. 3.11 Assignment of building blocks to placement requirements

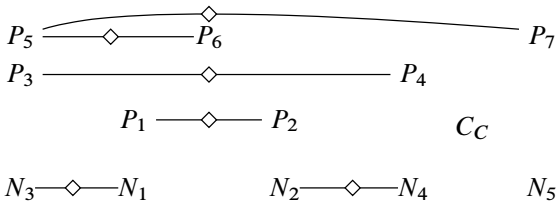


Fig. 3.12 Matching and proximity placement requirements from building blocks generated in the SMP graph for the circuit from Fig. 3.10

for the whole compound to reflect (3.13) and (3.14), which define the device location with respect to the axis coordinate c. By eliminating the coordinate c, a complete subgraph regarding the symmetry requirements is created.

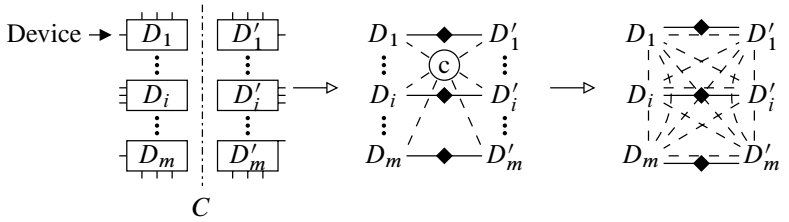


Fig. 3.13 Assignment of a symmetry compound C to matching and symmetry requirements

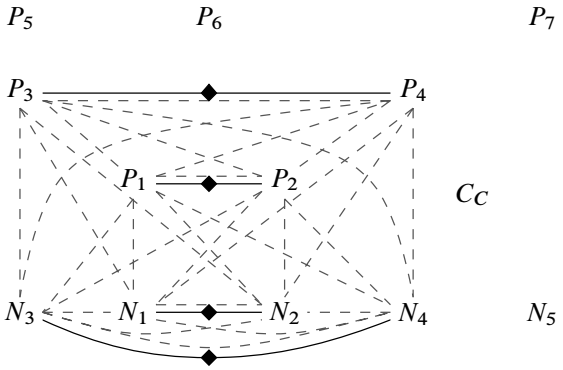


Fig. 3.14 Matching and symmetry requirements generated in the SMP graph for the circuit from Fig. 3.10

For the example circuit, the symmetry analysis algorithm determines four symmetric device pairs,

$$p_1 = (P_1, P_2), \qquad p_2 = (N_1, N_2), \qquad p_3 = (N_3, N_4), \qquad p_4 = (P_3, P_4),$$

and one symmetry compound

$$C_1 = \{p_1, p_2, p_3, p_4\},$$

which forms the set

$$S = \{C_1\}.$$

Figure 3.14 shows the additional edges generated in the SMP graph for matching and symmetry requirements. The final SMP graph containing all symmetry, matching and proximity requirements is depicted in Fig. 3.15.

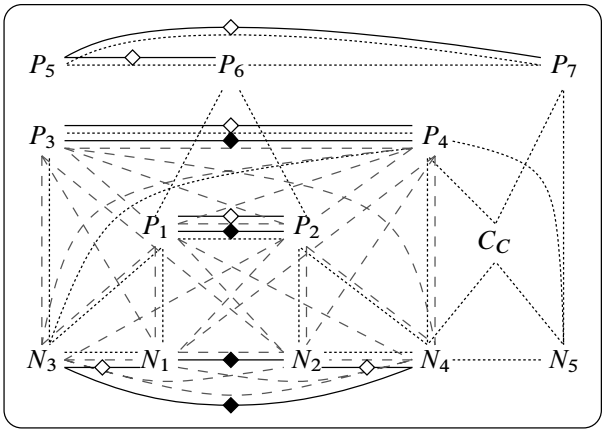


Fig. 3.15 SMP graph for the circuit from Fig. 3.10

3.2.3 HSMPG Tree and Its Generation

Definition 3.11 (HSMPG tree). An HSMPG tree is a hierarchical tree of symmetry, matching, and proximity groups. It describes the hierarchy of an analog circuit and placement requirements across the hierarchy. The leaf nodes of this tree are formed by the devices of the circuit and the inner nodes are called groups (see Definition 3.5). For every group, a type is defined that determines the placement requirements applying for its children. There are three different types of groups: A *proximity group (PG)* determines proximity requirements, a *matching group (MG)* determines matching requirements, and a *symmetry group (SG)* determines symmetry requirements.

3.2.3.1 Generation Algorithm

The SMP graph $\mathcal{G}_{\text{SMP}}$ together with the importance order relation T_I (3.19) are the basis to generate the HSMPG tree. The corresponding algorithm is denoted as Algorithm 3.1. Our method is similar to agglomerative methods known

Algorithm 3.1: Algorithm for HSMPG tree generation

Input: SMP graph $\mathcal{G}_{\text{SMP}}$, importance ordered type set T_I
Output: HSMPG tree

forall $\tau \in T_I$ **from** $\max(T_I)$ **to** $\min(T_I)$ **do**
  $\mathcal{G}_{\text{SMP}\,\tau} \leftarrow$ filter($\mathcal{G}_{\text{SMP}}$,τ);
  $\mathcal{G}_{\text{SMP}\,C} \leftarrow$ connectedComponents($\mathcal{G}_{\text{SMP}\,\tau}$);
  $\Gamma \leftarrow$createGroups($\mathcal{G}_{\text{SMP}\,C}$,$\tau$);
  $\mathcal{G}_{\text{SMP}} \leftarrow$buildSuperNodes($\mathcal{G}_{\text{SMP}}$,$\Gamma$);

from hierarchical cluster analysis [36]. Every group corresponds to a cluster and the similarity measure is given by the SMP graph $\mathcal{G}_{\mathrm{SMP}}$ and importance order relation T_I.

For each requirement type $\tau \in T_I$ in the importance order, first the subgraph $\mathcal{G}_{\mathrm{SMP}\tau}(\mathcal{N}_{\mathrm{SMP}\tau}, \mathcal{E}_{\mathrm{SMP}\tau})$ for this type is determined:

$$\mathcal{E}_{\mathrm{SMP}\tau} = \{e \in \mathcal{E}_{\mathrm{SMP}} \,|\, t_{\mathrm{SMP}}(e) = \tau\}\,, \tag{3.21}$$

$$\mathcal{N}_{\mathrm{SMP}\tau} = \{n \in \mathcal{N}_{\mathrm{SMP}} \,|\, \exists\,(e \in \mathcal{E}_{\mathrm{SMP}\tau}) : (n = e.a) \vee (n = e.b)\}\,. \tag{3.22}$$

It includes only edges $\mathcal{E}_{\mathrm{SMP}\tau}$ of type τ and the nodes they connect. Next, all connected components $\mathcal{G}_{\mathrm{SMP}C,i} \in \mathcal{G}_{\mathrm{SMP}C}$ of this graph are determined. A connected component is the largest subgraph $\mathcal{G}_{\mathrm{SMP}C,i}(\mathcal{N}_{\mathrm{SMP}C,i}, \mathcal{E}_{\mathrm{SMP}C,i})$, where every node $x \in \mathcal{N}_{\mathrm{SMP}C,i}$ can be reached from any other node $y \in \mathcal{N}_{\mathrm{SMP}C,i}$ [35].

For every connected component $\mathcal{G}_{\mathrm{SMP}C,i}(\mathcal{N}_{\mathrm{SMP}C,i}, \mathcal{E}_{\mathrm{SMP}C,i})$ that has more than one node, i.e., $|\mathcal{N}_{\mathrm{SMP}C,i}| > 1$, a new group $\gamma \in \Gamma$ is created: The nodes $\mathcal{N}_{\mathrm{SMP}C,i}$ form the children of the new group. The type of the group is determined by τ. If $\tau \in \{M_{\mathrm{B}}, M_{\mathrm{S}}\}$, then a matching group is created. If $\tau = S$, then a symmetry group is created. Otherwise, the new group will be a proximity group.

Finally, a super node S in G_R is formed for every group $\gamma \in \Gamma$. Edges $e \in \mathcal{E}_{\mathrm{SMP}}$ which would connect nodes inside and outside the super node are replaced by edges that refer to the super node.

3.2.3.2 Example

Figure 3.16 shows how the SMP graph (Fig. 3.15) of the example circuit (Fig. 3.10) is processed. The resulting HSMPG tree is shown in Fig. 3.16h. In the first iteration of the algorithm, the matching requirements M_{S} are evaluated (Fig. 3.16a) and the matching groups $\mathrm{MG}_{S,1}$ to $\mathrm{MG}_{S,4}$ are created (Fig. 3.16b). For each group, a super node is formed and the requirement edges are transformed to refer to the new super nodes (Fig. 3.16c). Next, requirements of type M_{B} are handled. In Fig. 3.16c, there are three such requirements between the groups $\mathrm{MG}_{S,2}$ and $\mathrm{MG}_{S,3}$, between the devices P_5 and P_6 and between the devices P_5 and P_7. $\mathrm{MG}_{S,2}$ and $\mathrm{MG}_{S,3}$ form the new group $\mathrm{MG}_{B,1}$. The devices P_5, P_6, and P_7 are all part of the same connected component and form the group $\mathrm{MG}_{B,2}$ (Fig. 3.16d). The further processing leads to the creation of symmetry group SG_1 (Fig. 3.16e, f), representing the symmetry compound. Proximity group $\mathrm{PG}_{N,1}$ is created because of the proximity requirements from the netlist and represents the complete circuit (Fig. 3.16g, h).

3.2.3.3 Discussion

Our approach of a static importance order of the requirement types has led to correct results for all our experiments. Nevertheless, a dynamic approach is also possible. For example, the influence of a constraint violation to the circuit performances

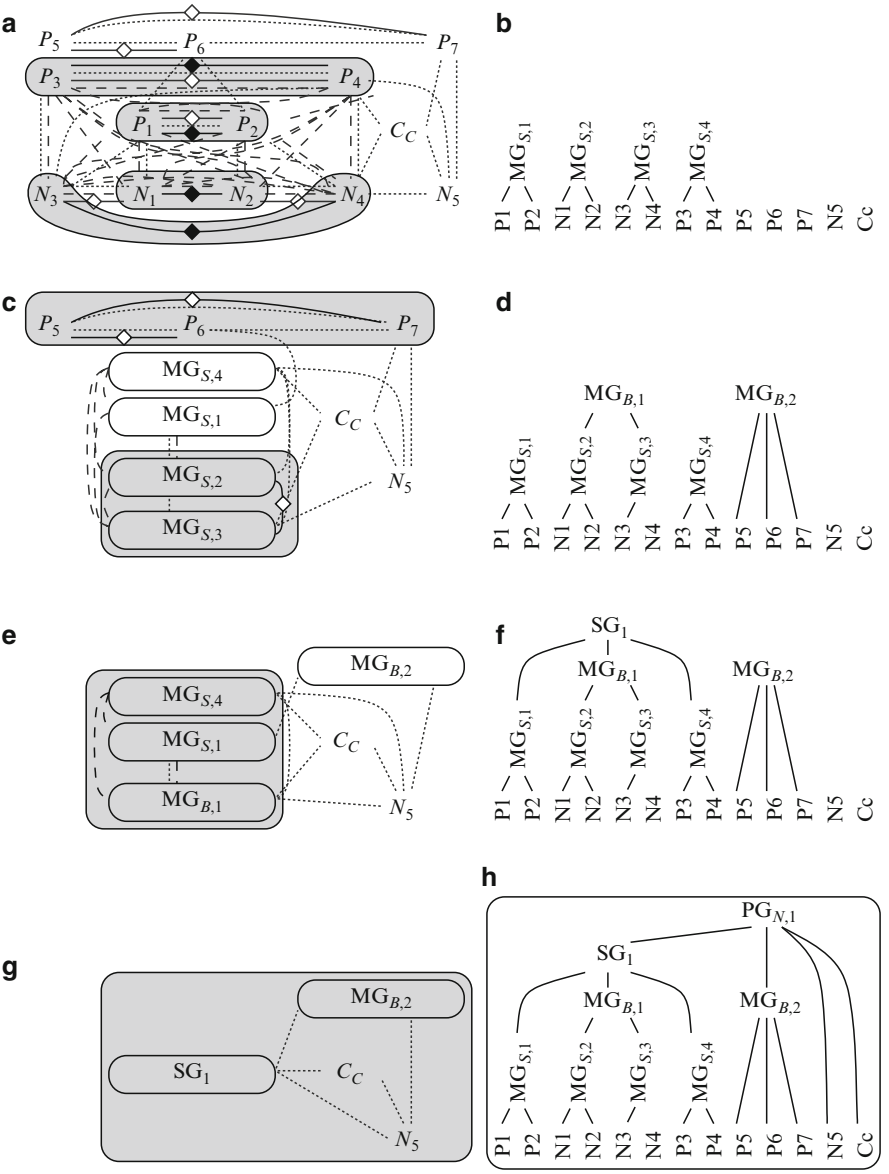


Fig. 3.16 Iteration steps for the circuit from Fig. 3.10 and the SMP graph from Fig. 3.15. SMP graphs and HSMPG trees for iteration M_S (**a, b**), iteration M_B (**c, d**), iteration S (**e, f**), and iteration P_N (**g**). The final HSMPG tree is shown in (**h**)

could be determined by simulation. The result could be used to determine a priority function $\Phi : \mathcal{E}_{\text{SMP}} \to \mathbb{N}$. This function can then be used to determine the order in which the constraint requirements are processed.

3.2.4 Constraint Generation

Besides information about the hierarchy of a circuit, the generated HSMPG tree contains information about the constraints that have to be applied within each hierarchical group and among hierarchical groups. This information has to be formatted in the placer-specific constraint input format, including a constraint generation according to Sect. 3.1.2.

Same layout variants and alignment constraints are generated for matching groups. In addition, an inherent proximity constraint is valid because all devices to match are in the same group. If the devices in the circuit consist of a number of subdevices instead of a large single part (e.g., a transistor realized as parallel subtransistors), then the alignment constraint is replaced by a common centroid constraint.

For symmetry groups, the computed hierarchy is exploited to generate constraints implementing (3.13) and (3.14). Inside each M_S-matching group, the centers of the devices are aligned in the direction of the symmetry axis. For all groups belonging to the same axis, a group alignment constraint is generated to align their centers perpendicular to the symmetry axis. For example, four module-related constraints would be created for the four symmetry pairs of the example circuit and a vertical axis:

$$y_{P1} = y_{P2} \qquad y_{N1} = y_{N2} \qquad y_{N3} = y_{N4} \qquad y_{P3} = y_{P4}. \tag{3.23}$$

This constraint is denoted as *symmetry (pair)*. For each of the matching groups $\mathrm{MG}_{S,1}$ to $\mathrm{MG}_{S,4}$ the centers in x-direction $x_{\mathrm{MG}_{S,i}}$ are calculated by

$$x_{\mathrm{MG}_{S,i}} = \frac{1}{2}(x_{m_1} + x_{m_2}), \tag{3.24}$$

where m_1 and m_2 are the modules of each group. These groups are then aligned in horizontal direction:

$$x_{\mathrm{MG}_{S,1}} = x_{\mathrm{MG}_{S,2}} = x_{\mathrm{MG}_{S,3}} = x_{\mathrm{MG}_{S,4}} \tag{3.25}$$

This constraint is denoted as *symmetry (groups)*. Overall, (3.13) and (3.14) are implemented by (3.23)–(3.25) for this circuit.

The placement algorithm presented in the following constructs the layout bottom up using the hierarchy given by the HSMPG tree. It inherently keeps the elements of each group in close spatial proximity. Therefore, no explicit proximity constraints have to be formulated for proximity groups.

3.3 B*-Tree Placement Considering Linear and Piecewise-Linear Constraints

The deterministic placer Plantage generates placements using a hierarchically bounded enumeration. During the enumeration process, many different B*-trees have to be evaluated for parts of the circuit as well as for the whole circuit. The B*-trees are evaluated based on the corresponding placements. The proposed methodology is used to generate placements with respect to arbitrary linear as well as piecewise-linear constraints from a feasible B*-tree. Linear constraints are needed for symmetry and common centroid constraints, as well as for linear minimum distance constraints. Piecewise-linear constraints are needed for minimum distance constraints of special devices, such as DTI transistors (see Sect. 3.1.2).

Horizontal and vertical relationships between modules are modeled by two directed constraint graphs, HCG $= (\mathcal{N}_h, \mathcal{E}_h)$, and VCG $= (\mathcal{N}_v, \mathcal{E}_v)$. HCG and VCG both consist of a set of nodes $\mathcal{N}_h$, and $\mathcal{N}_v$, and a set of directed edges $\mathcal{E}_h$, and $\mathcal{E}_v$. Any directed edge is an ordered pair of nodes. In this approach, each module has a corresponding node in HCG as well as in VCG. A directed edge $e \in \mathcal{E}_h$, $e = (n_i, n_j)$, denotes that module i has to be placed left of j. A directed edge $e \in \mathcal{E}_v$, $e = (n_i, n_j)$, denotes that module i has to be placed below j.

In Fig. 3.17, an overview of the methodology is shown. Algorithm 3.2 generates the VCG for the given B*-tree. An edge (n_i, n_j) in VCG requires module j to be placed above i. Thus, an inequality $y_i + h_i < y_j$ is formulated. For the example in Fig. 3.5, the VCG is shown in Fig. 3.18. According to this VCG, the following inequalities are formulated: $y_a \geq y_s$, $y_b \geq y_a + h_a$, $y_c \geq y_s$, $y_d \geq y_c + h_c$, $y_e \geq y_b + h_b$, $y_e \geq y_d + h_d$.

First, the methodology is explained in the next section for linear constraints only. It can be solved by a linear program (LP), minimizing the height of the placement

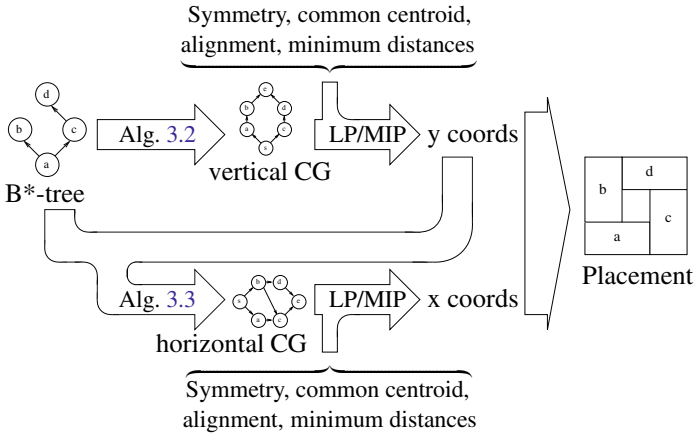


Fig. 3.17 Placement generation from a B*-tree considering constraints

Algorithm 3.2: buildVCG(VCGNode thisNode, predecessor)
(© IEEE 2008, [37])

```
begin
    if B*-tree node of thisNode has a left child then
        leftNode ← new VCG node for left child;
        add edge from thisNode to leftNode;
        buildVCG(leftNode, thisNode);
    else
        add edge from thisNode to the end node;
    if B*-tree node of thisNode has a right child then
        rightNode ← new VCG node for right child;
        add edge from predecessor to rightNode;
        buildVCG(rightNode, predecessor);
end
```

subject to symmetry, common centroid, and minimum distance constraints. The results of the LP are the y coordinates of the modules. An approach to handle piecewise-linear constraints is described later in Sect. 3.3.2.

3.3.1 Linear Constraint Handling

The vertical constraint graph is built as described in Algorithm 3.2: if module i is a left child of module j in the B*-tree, then n_i is the direct successor of n_j in the CG. If module i is a right child of module j in the B*-tree, then n_i and n_j share the same predecessor. At the beginning of Algorithm 3.2, a start node is created. Also, a node corresponding to the root node of the B*-tree is added with the start node being its predecessor. Both nodes are passed to the algorithm.

A linear program is formulated using VCG:

$$\mathbf{y}_{\text{opt}} = \arg\min_{\mathbf{y}} y_e, \tag{3.26}$$

$$\text{s.t.} \quad \underbrace{\mathbf{M}_v \cdot \mathbf{y} \geq \mathbf{d}_v}_{\text{Minimum distance constraints}}, \tag{3.27}$$

$$\underbrace{\mathbf{C}_v \cdot \mathbf{y} = \mathbf{k}_v}_{\text{Symmetry \& common centroid constraints}}. \tag{3.28}$$

The vector $\mathbf{y}$ is the vector of y coordinates for all modules and y_e is the y coordinate of the virtual end node. The matrix $\mathbf{M}_v$, together with $\mathbf{d}_v$, defines the minimum vertical distances between the modules. The matrix $\mathbf{C}_v$, together with the vector $\mathbf{k}_v$, defines the symmetry and common centroid constraints for the vertical axis. Minimizing y_e is equivalent to minimizing the total height of the placement. The vector $\mathbf{y}_{\text{opt}}$ represents the optimal y coordinates for the given B*-tree.

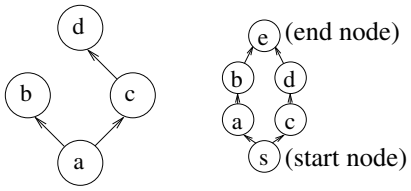


Fig. 3.18 Example: B*-tree and its corresponding VCG

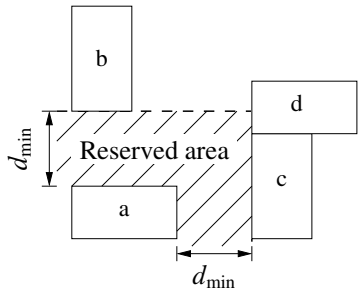


Fig. 3.19 Expected placement for the B*-tree of Fig. 3.18 with additional minimum distance constraints

An example placement for the B*-tree of Fig. 3.18 with the minimum distance constraints

$$\underset{n \neq a}{\forall} d(a,n) \geq d_{\min} \tag{3.29}$$

is shown in Fig. 3.19. After the computation of the y coordinates, the minimum distance constraints are fulfilled for the y-axis for those modules, which have a connecting edge in VCG. To ensure that all constraints are fulfilled, two cases need to be considered for pairs of modules having a minimum distance constraint during the generation of HCG:

1. If the y projections of the two modules are overlapping, an edge is created in HCG between these two modules. The weight of that edge must be greater than or equal to the minimum distance between the two modules.
2. If the y projections of the two modules are not overlapping, it must be checked if their distance in vertical direction is sufficient to fulfill the minimum distance constraint. If not, an extra edge has to be created to ensure the minimum distance.

To handle both cases efficiently, a *shadowing* algorithm is proposed to generate HCG. During the generation of HCG, a module can cast a *core shadow* as well as a *partial shadow*.

Definition 3.12 (Core shadow). The region on the y-axis, which is covered by module m is called its core shadow:

$$\Upsilon_{\mathrm{CS},m} = [y_m; y_m + h_m]. \tag{3.30}$$

The lower y-coordinate and height of module m are denoted by y_m and h_m, respectively.

Definition 3.13 (Partial shadow). The partial shadow $\Upsilon_{\mathrm{PS},m}$ that module m casts on the y-axis is defined as:

$$\Upsilon_{\mathrm{PS},m} = [y_m - d_{\max}(m); y_m] \cup, \tag{3.31}$$

$$[y_m + h_m; y_m + h_m + d_{\max}(m)]. \tag{3.32}$$

The region of the y-axis, where modules may be influenced by the placement of module m, is covered by the partial shadow of m. The shadows of a module can be illustrated as shown in Fig. 3.20.

To efficiently build HCG, y-regions are defined. Therefore, all lower and upper y-coordinates y_m and $y_m + h_m$ of all modules m are stored in a list, sorted, and unified. Any region between two entries of that list is called a y-region. A tree data structure is used to store the sets of modules being associated with the y-regions. This allows for more than one module to be registered with a single region. Using these definitions, the shadowing algorithm can be formulated as in Algorithm 3.3. The algorithm only creates edges, which are required to keep the minimum distances.

For the example in Fig. 3.19, the algorithm is demonstrated in Fig. 3.21. The regions are initialized in Fig. 3.21a. Then, module a is registered in Fig. 3.21b. The core shadow of a overlaps the lowest region, the partial shadow overlaps the two regions above. Thus, a is registered with all of these three regions. A HCG edge is created from the virtual start node to node of module a. Module b is registered with the regions of its core shadow in Fig. 3.21c. The partial shadow of b is overlapping regions, where a is already registered. Thus, b is appended to the list of registered modules in those regions, and an edge from the start node to the HCG node of module b is created. In Fig. 3.21d, module c overwrites the lower two regions. Since a has been overwritten, a HCG edge is created. There is no edge created from b to c, although b has been overwritten as well. This is because there is no minimum distance constraint defined between these two modules. Finally, in Fig. 3.21e, module d overwrites regions, where a, b, and c have been registered. An edge from b to d is created, and, due to the minimum distance constraints, an extra edge is created from a to d. Figure 3.21f shows the complete HCG along with the corresponding valid placement.

Using HCG, the optimization problem can be formulated as a minimization of the x-coordinate of the virtual end node, x_e, with respect to all constraints:

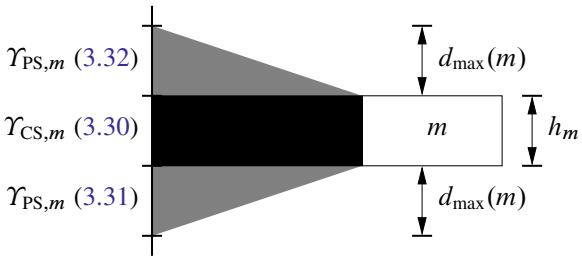


Fig. 3.20 Core shadow $\Upsilon_{\mathrm{CS},m}$ and partial shadow $\Upsilon_{\mathrm{PS},m}$ of module m

Algorithm 3.3: buildHCG()

```
begin
    startNode ← new HCG node as start;
    endNode ← new HCG node as end;
    create tree of y-regions;
    initialize all y-regions with startNode;
    forall modules m in preorder do
        modNode ← new HCG node for m;
        forall regions r in Υ_CS,m do
            forall modules n registered in r do
                if d_vert(m, n) < d_min(m, n) then
                    add edge to modNode from HCG node of n;
            remove all entries in r;
            register module in r;
        forall regions r in Υ_PS,m do
            register module in r;
    remove multiple edges;
    add edges from all nodes in region list to endNode;
end
```

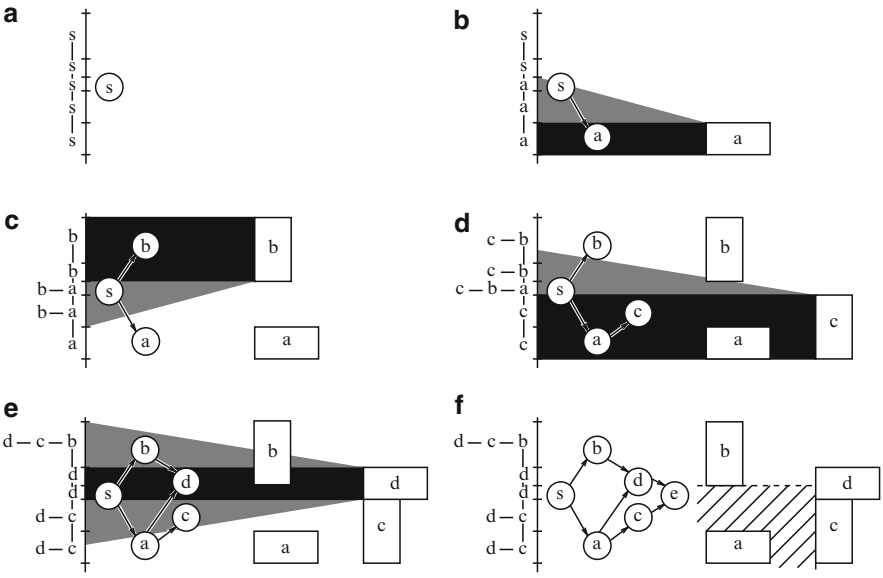


Fig. 3.21 New approach for building HCG with minimum distance constraints (**a**)–(**e**), and the complete HCG in (**f**)

$$\mathbf{x}_{\text{opt}} = \arg\min_{\mathbf{x}} x_e \tag{3.33}$$

$$\text{s.t.} \quad \underbrace{\mathbf{M}_h \cdot \mathbf{x} \geq \mathbf{d}_x}_{\text{Minimum distance constraints}} , \tag{3.34}$$

$$\underbrace{\mathbf{C}_h \cdot \mathbf{x} = \mathbf{k}_h}_{\text{Symmetry \& common-centroid constraints}} \tag{3.35}$$

This optimization problem can be solved by a Simplex solver. The vector $\mathbf{x}_{\text{opt}}$ then contains the optimal x coordinates of all modules.

3.3.2 *Piecewise-Linear Constraint Handling*

As an example for piecewise-linear constraints, the allowable ranges for the distance between DTI transistors are shown in Fig. 3.4. Since there is a forbidden zone between two allowable ranges, the solution space is concave. Due to this fact, the problem can no longer be formulated as a linear programming problem.

To generate placements subject to piecewise-linear minimum distance constraints, a formulation of the problem is proposed, which can be solved by a linear mixed integer programming (MIP) solver. W.l.o.g., the proposed approach is described for HCGs and for the piecewise-linear constraints for DTI transistors, as defined in Sect. 3.1.2. The same methodology is applied when solving for the vertical coordinates.

In a constraint graph, a module is represented by a node. In general, there is at least one edge ending at the node, and at least one edge starting from the node. The weight of an edge represents the distance between the modules of the nodes being connected. An example is depicted by Fig. 3.22.

The distance between two modules subject to the piecewise-linear minimum distance constraint (3.18) can be described by the two allowable ranges. For every edge, a binary *range variable* is defined to indicate in which range the distance is located. For an edge e_i, the range variable is r_{ei}.

$$r_{ei} \in \{0, 1\}. \tag{3.36}$$

An equation $r_{ei} = 0$ indicates that the weight of e_i is in Range 1, $r_{ei} = 1$ indicates that the weight of e_i is in Range 3.

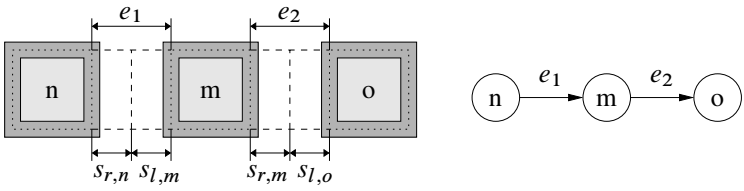


Fig. 3.22 Example with three modules and a corresponding HCG. The stretch variables $s_{r,n}$, $s_{l,m}$, $s_{r,m}$, and $s_{l,o}$ are only valid in case the trenches are shared

The following equations and inequalities are formulated for e_1 in Fig. 3.22. For e_2, similar constraints need to be defined. Together with the range variables, two inequalities are formulated to fulfill (3.18) with β being a sufficiently large number:

$$e_1 - r_{e1} \cdot \beta \leq s_{\max,e1} - s_{r,m}, \tag{3.37}$$

$$e_1 + (1 - r_{e1}) \cdot \beta \geq d_{\text{DTI}}, \tag{3.38}$$

$$e_1 \geq s_{r,n} + s_{l,m}. \tag{3.39}$$

Equations (3.37) and (3.38) include two cases:

1. For $r_{e1} = 0$: Here (3.37) reduces to $e_1 \leq s_{\max,e1} - s_{r,m}$, and (3.38) reduces to $e_1 + \beta \geq d_{\text{DTI}}$. Since e_1 is positive, and with β being greater than d_{DTI}, inequality (3.38) is always fulfilled. Thus, (3.37) makes sure that e_1 cannot exceed its maximum stretching value $s_{\max,e1}$ minus the stretching of module m to the right, $s_{r,m}$.
2. For $r_{e1} = 1$: Here (3.37) reduces to $e_1 - \beta \leq s_{\max,e1} - s_{r,m}$, and (3.38) reduces to $e_1 \geq d_{\text{DTI}}$. With β being a sufficiently large number (3.37) is always fulfilled. In this case, the distance between n and m must be at least d_{DTI}.

To make sure that the distance between the modules n and m is always sufficient to generate the DTI, (3.39) is formulated. It can be shown that it is sufficient to set β to a value greater than the sum of all module widths plus their worst-case minimum distances (times the number of symmetry constraints +1). An edge weight cannot exceed this limit.

For the case of DTI transistors, an additional constraint needs to be considered: As described in Sect. 3.1.2, the trenches surrounding a module m can be stretched to the left ($s_{l,m}$) and to the right ($s_{r,m}$), up to a certain limit of

$$s_{l,m} + s_{r,m} \leq s_{\max,m}. \tag{3.40}$$

The stretching variables $s_{l,m}$ and $s_{r,m}$ can be calculated from e_1 and e_2, respectively:

$$e_1 - s_{l,m} - s_{r,n} - r_{e1} \cdot \beta \leq 0. \tag{3.41}$$

$$e_2 - s_{r,m} - s_{l,o} - r_{e2} \cdot \beta \leq 0. \tag{3.42}$$

In case e_1 is in range 1 ($r_{e1} = 0$), the trenches are stretched. Hence, e_1 must be equal to $s_{l,m} + s_{r,n}$. This is made sure by (3.41) if the stretching variables are secondarily minimized due to a term in the cost function. For range 3, (3.41) is always fulfilled because of the β term. Equation (3.42) can be explained similarly.

To solve for the x-coordinates, the optimization problem (3.33)–(3.35) needs to be extended as follows: For every edge i, an additional range variable $r_{ei} \in \{0, 1\}$ and constraints similar to (3.37)–(3.39), (3.41), and (3.42) need to be defined. For every module m, a constraint similar to (3.40) needs to be added. Furthermore,

the cost function (3.33) is extended by a λ term, secondarily minimizing the total stretchings of all modules:

$$x_e + \lambda \cdot \sum_m (s_{l,m} + s_{r,m}). \tag{3.43}$$

This formulation allows for the use of a linear MIP solver. The factor λ must be sufficiently small to make sure that x_e is the main minimization objective. For DTI transistors, it is sufficient to set λ to a positive value of less than $\frac{w_{\min}}{N \cdot d_{\mathrm{DTI}}}$, with $w_{\min}$ being the width of the smallest module, and N being the total number of modules.

3.4 Enhanced Shape Functions

To handle the combination of different partial placements in an efficient, area-saving way, enhanced shape functions are introduced in this section. First, a brief review of standard shape functions is given. Then, enhanced shape functions and the enhanced combination of shapes are described in the following.

3.4.1 Review of Shape Functions

Shape functions [38] can be used to calculate compact placements for a set of rectangular modules. A shape function is defined as an ordered set of shapes. Each shape represents a placement with a different aspect ratio. Therefore, a shape describes one possible placement of a module set by its bounding rectangle size, which is formulated as a tuple (w, h), where w and h denote the width and height, respectively.

In order to generate placements, a recursive algorithm is defined to calculate the shape function of the module set: First, the set of modules is partitioned into two subsets. For each subset, a shape function is calculated, and the shape functions are then combined to generate a shape function for the complete module set. If a subset consists of only a single module, the shape function only consists of a single shape, representing the width and the height of this module. To combine two shape functions, all possible combinations of the shapes of both shape functions are evaluated. This can be done using a fast operation. Since shapes represent the bounding rectangles of their corresponding placements, combining two shapes means calculating a common bounding rectangle for the two corresponding placements. Two placements can be combined either horizontally or vertically. For shapes, this is called horizontal and vertical addition. The result of a horizontal addition of two shapes (w_1, h_1) and (w_2, h_2) is $(w_1 + w_2, \max(h_1, h_2))$. An example of horizontal addition is shown in Fig. 3.23. A vertical addition results in a shape $(\max(w_1, w_2), h_1 + h_2)$. An example of a shape function is shown in Fig. 3.24. There, all combinations have been evaluated. In this diagram, there are suboptimal

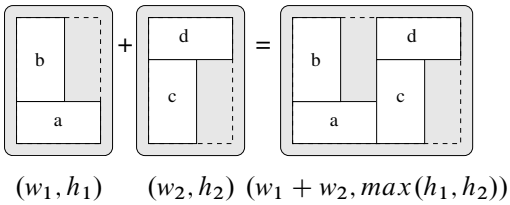


Fig. 3.23 Standard shape addition ([37], © IEEE 2008)

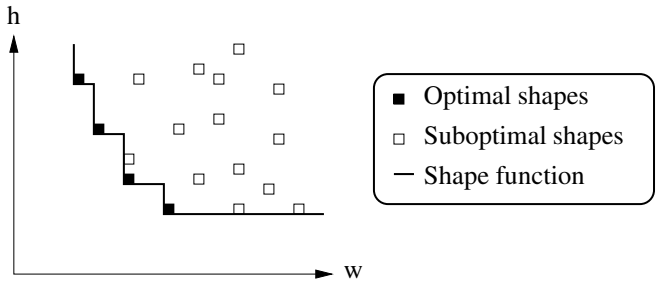


Fig. 3.24 All shapes of a resulting shape function (SF)

shapes which have a bigger height than other shapes having the same or even a lower width. These suboptimal shapes are removed before further calculations are performed. Removing suboptimal shapes significantly reduces the time needed in subsequent steps, while the quality of the solution remains unchanged. This can be considered to be a key feature of shape functions. Based on the remaining shapes, a continuous shape function can be drawn, as shown in Fig. 3.24. This continuous shape function can be considered the Pareto front of possible placements.

When the recursive algorithm has terminated, there is a shape function for the entire circuit. This shape function represents different placements, having different aspect ratios. This is a second key feature of shape functions.

3.4.2 Definition of Enhanced Shape Functions

The corresponding placement of a shape can be described as a slicing tree, since it is built by horizontal and vertical additions of other placements [39]. Nonslicing placements cannot be handled by a slicing tree. Since the solution space is limited by this fact, the solution quality may be degraded.

In this section, enhanced shape functions [37] are described, which preserve the key features of shape functions while at the same time being able to handle non-slicing placements. An enhanced shape is defined as (w, h, α). The corresponding B*-tree α of a placement is stored in addition to the placement's bounding box (w, h). In this chapter, B*-trees are denoted by Greek lowercase letters. Storing the B*-tree allows for efficient combination of the enhanced shape functions and their

underlying modules, as described in the subsequent section. The widths and heights of the enhanced shapes are used to calculate the Pareto front, and to identify the suboptimal enhanced shapes to be removed. An enhanced shape is considered to be suboptimal in two cases:

- The enhanced shape has a bigger height than other enhanced shapes having the same or even lower width.
- The enhanced shape has a higher netlength than other enhanced shapes, having the same width and height.

3.4.3 Combination of Enhanced Shape Functions

To combine two enhanced shape functions, all of their enhanced shapes are combined in pairs. In contrast to standard shapes, the combination of two enhanced shapes (w_i, h_i, α) and (w_j, h_j, β) is calculated using the B*-trees. The widths w_i, w_j and the heights h_i, h_j can be used to estimate the resulting enhanced shape. For a horizontal addition, an upper bound for the size of the resulting placement can be defined as $(w_i + w_j, \max(h_i, h_j))$. For a vertical addition, the upper bound can be defined as $(\max(w_i, w_j), h_i + h_j)$. Using the B*-trees α and β, the size of the resulting placement can be smaller than $(w_i + w_j, \max(h_i, h_j))$ and $(\max(w_i, w_j), h_i + h_j)$, respectively.

Figures 3.23 and 3.25 show the differences between the horizontal addition of conventional and enhanced shapes for a simple example. It is obvious that $w_1 + w_2$ is greater than w_{sum}. Generally speaking, more compact placements can be reached if the enhanced shape function combination is used.

Two methods are proposed to add the B*-trees of enhanced shapes horizontally and vertically. They are described in the following paragraphs. For both methods, it is shown that the outcome of adding two feasible B*-trees is also a feasible B*-tree. This is an important property of the addition operations. Due to that property, the algorithm avoids calculations for many infeasible B*-trees.

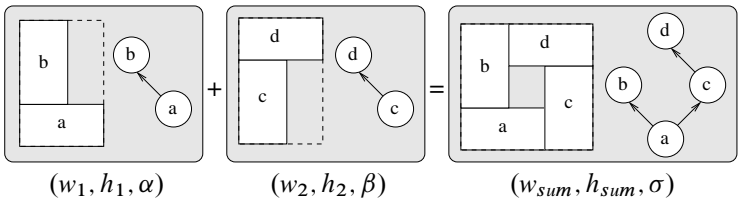


Fig. 3.25 Horizontal enhanced shape addition ([37], © IEEE 2008)

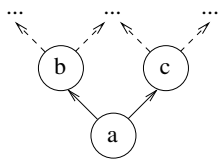


Fig. 3.26 An arbitrary B*-tree to define the in- and preorder traversal

Horizontal Addition

For two given B*-trees α and β, a horizontal addition is performed by attaching the root node of β to the *lowest, rightmost* node of α. The lowest, rightmost node of a B*-tree is defined as the node with no right child, while this node itself and all its predecessors are either right children or the root node.

Due to the characteristics of the placement algorithm for B*-trees, the resulting placement is compact to the lower left corner. Horizontal addition has a notable property. Any constraint, which was satisfied by α and β before the addition is also satisfied by the resulting B*-tree. This holds true for as long as no additional constraints apply for the superset of the modules of the two B*-trees. This property can be derived from the in- and preorder traversals of the B*-trees, because they can be used to determine the feasibility of constraints (see Chap. 1). As shown in Fig. 3.26, the in- and preorder traversals of an arbitrary B*-tree, are defined as ordered lists recursively by the following equations:

$$\text{in}(a) := \text{in}(b), a, \text{in}(c) \tag{3.44}$$

$$\text{pre}(a) := a, \text{pre}(b), \text{pre}(c). \tag{3.45}$$

Two order relations can be defined on the traversals:

- $a \overset{\text{IN}}{\prec} b$ means "a is a predecessor of b in the inorder traversal."
- $a \overset{\text{PRE}}{\prec} b$ means "a is a predecessor of b in the preorder traversal."

The topologies of the B*-trees α and β are not changed by the horizontal addition. There is only one edge added to connect the two B*-trees to generate B*-trees σ. The feasibility of the constraints can be checked using the relative positions of the modules in the in- and preorder traversals [22]. Without loss of generality, the node c can be considered the root node of β, and a the lowest, rightmost node of α. Considering this fact together with (3.44) and (3.45), the relative positions of the modules in α and β in the in- and preorder traversals do not change. Thus, any constraint, which was satisfied before, is also satisfied after the addition of the two B*-trees. The example given by Fig. 3.25 illustrates a horizontal addition.

Vertical Addition

A vertical addition is intended to arrange two partial placements vertically, generating compact results. In Fig. 3.27, a compact result of a vertical addition is shown

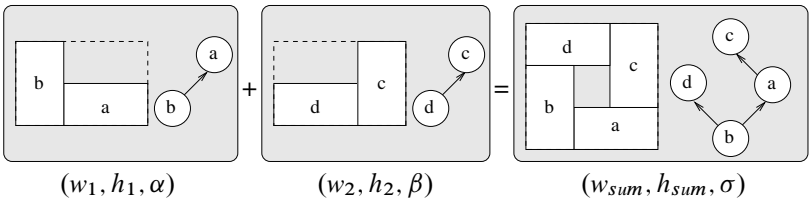


Fig. 3.27 Vertical enhanced shape addition ([37], © IEEE 2008)

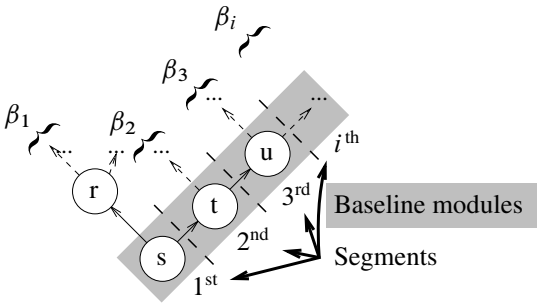


Fig. 3.28 Baseline modules and segments of a B*-tree β ([37], © IEEE 2008)

together with its corresponding B*-tree σ. In contrast to the horizontal addition, the resulting B*-tree σ cannot be generated easily by adding an edge from α to β. For this reason, an algorithm is proposed, which iteratively forms a new B*-tree as the result of the addition of α and β. The topology is changed to achieve better placements.

In a B*-tree, all modules, which can be reached starting from the root node traversing right edges only are placed close to the baseline. These modules are denoted as *baseline modules* in this chapter. The B*-tree β is segmented in the proposed approach, with the root node of each segment being a baseline module. In Fig. 3.28, the baseline modules and the segments of a B*-tree are depicted. Furthermore, adequate nodes of α are then determined, which serve as the new parents of the segment root nodes. This is done using a contour-based algorithm, which assigns one segment after the other, from "left to right," in the order given in Fig. 3.28.

Figure 3.29 illustrates how the B*-tree σ of Fig. 3.27 was built by the algorithm. First, no segment of β is added to α, as shown in Fig. 3.29a. A contour is drawn as a thick line above the placement for α. After that, the first segment of β, representing the node d, is added in Fig. 3.29b. The projection of d on the x-axis shadows b and parts of a. Thus, a and b are both potential parent nodes for d. Node d is added as a left child of module b, because b limits the y-coordinate of d when shifting it downward. According to the same rules, c is appended as a left child of a, shown in Fig. 3.29c.

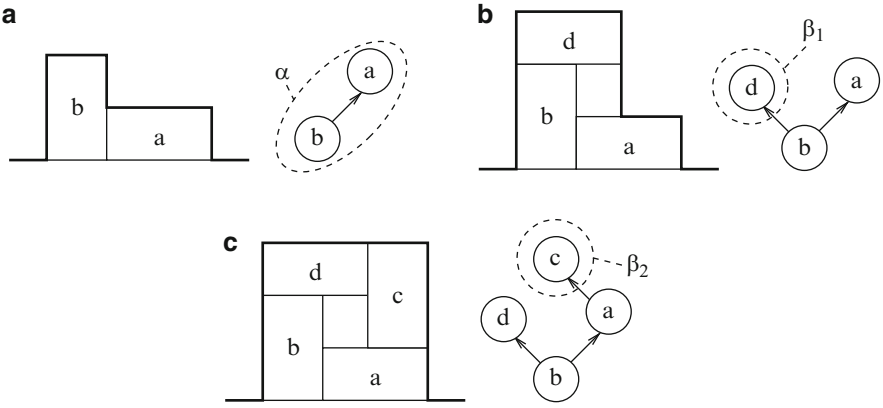


Fig. 3.29 Iteratively changing the B*-tree topology for a vertical addition ([37], © IEEE 2008), the initial B*-tree α (**a**), adding the segments of β in (**b**) and (**c**)

If the vertical addition is performed in this way, the constraints remain satisfied: The ith and $(i+1)$th segments of β are denoted as β_i and β_{i+1}, and root(β_i) denotes the root node of β_i. The segments of the B*-tree β are added in ascending order, as defined in Fig. 3.28. Before the addition, the in- and preorder positions of the segments fulfill the following conditions:

$$\text{in}(\text{root}(\beta_i)) \overset{\text{IN}}{\prec} \text{in}(\text{root}(\beta_{i+1})), \tag{3.46}$$

and

$$\text{pre}(\text{root}(\beta_i)) \overset{\text{PRE}}{\prec} \text{pre}(\text{root}(\beta_{i+1})). \tag{3.47}$$

The root node of β_i is then added to a node of α, denoted as parent(β_i), being its left child. The segments are added "from left to right". Therefore, the parents fulfill the condition

$$\text{parent}(\beta_i) \overset{\text{IN}}{\prec} \text{parent}(\beta_{i+1}). \tag{3.48}$$

and

$$\text{parent}(\beta_i) \overset{\text{PRE}}{\prec} \text{parent}(\beta_{i+1}). \tag{3.49}$$

Thus, the relative positions of the modules in the in- and preorder traversals of α, β_i and β_{i+1} do not change. Consequently, the feasibility of the constraints remains unchanged. Similar to the horizontal addition, this holds true for as long as no additional constraints apply for the superset of the modules of the two B*-trees.

Using this approach, the key advantages of standard shape functions are maintained. All suboptimal enhanced shapes are stripped after the addition of two enhanced shape functions. This reduces the computational effort in subsequent steps efficiently. Furthermore, a set of possible placements is stored, instead of a single solution.

3.5 Hierarchically Guided Enumeration

It is obvious that the enumeration of the complete solution space yields the optimal result. However, this cannot be performed for most circuits because of long run times. This becomes clear when considering the number of different B*-trees for n modules [22]. There are 336 different B*-trees for four modules, while for eight modules, there are 57, 657, 600 different B*-trees. Thus, a complete enumeration is impossible in practical cases. As a consequence, the presented enumeration approach is guided by the HSMPG tree, as described in Sect. 3.2, to limit the number of elements, which are considered in an enumeration run. The hierarchy can be illustrated as a hierarchy tree (see Sect. 3.2.3), where the root represents the whole circuit. The leaf nodes represent the modules, their parents represent analog structures, such as differential pairs (DP) or current mirrors (CM). Figure 3.30 shows a typical schematic of a Miller operational amplifier, together with its HSMPG tree, which was automatically generated from the netlist.

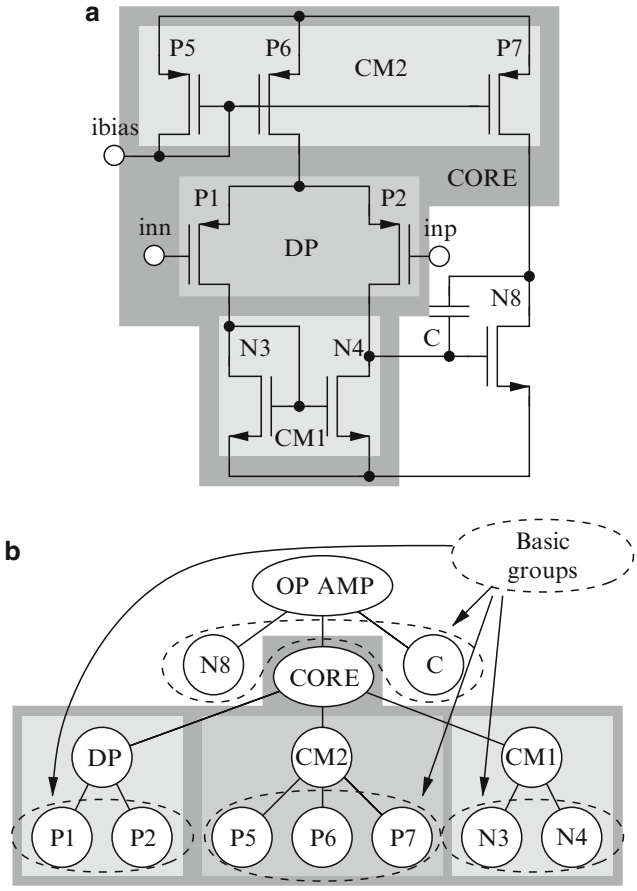


Fig. 3.30 Miller op amp schematic (**a**) and HSMPG tree (**b**) ([37], © IEEE 2008)

The HSMPG tree is used to perform a bottom-up enumeration. First, all possible placements for the basic groups are evaluated. These basic groups are formed by the modules of leaf nodes having the same parent node in the hierarchy tree. In the given example of Fig. 3.30, these sets are {P1, P2}, {N3, N4}, {P5, P6, P7}, and {C, N8}.

The enumeration of all possible placements of a basic group is done by evaluating all possible B*-trees for the modules of this set. Considering the variant matching constraints, the allowed combinations of variants are enumerated. This procedure is called *basic enumeration* in this chapter and is depicted in Fig. 3.31. For all feasible B*-trees, the basic enumeration evaluates all possible variants of the modules. That means, e.g., it evaluates different numbers of fingers for a transistor, or different aspect ratios for a capacitor. During the enumeration, variant constraints are considered. The result of the basic enumeration is an enhanced shape function for the basic group. The enhanced shape function only stores the placements, which potentially contribute to a good result for the whole circuit. The basic enumerations can be parallelized easily, since they are independent of each other.

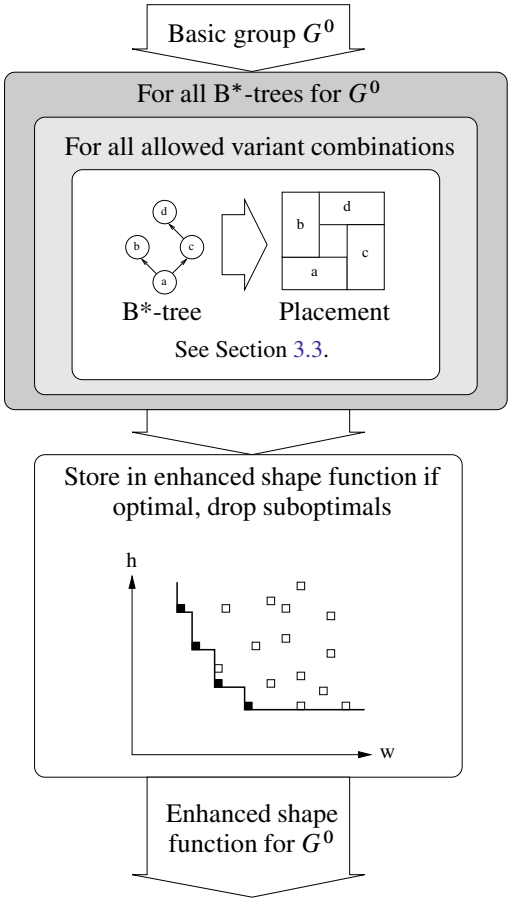


Fig. 3.31 The basic enumeration

After Plantage has determined the enhanced shape functions of all basic groups, the algorithm steps up to the next level of hierarchy (defined by the HSMPG tree). At this level, the enhanced shape functions of the basic groups are combined in every possible sequence (see Sect. 3.4.3). The algorithm terminates as soon as the enhanced shape function for the complete circuit has been calculated. This is the key algorithm in this approach, which is depicted in Fig. 3.32 and described in Algorithm 3.4. In the example given by Fig. 3.30, the algorithm combines the enhanced shape functions of the differential pair DP and the two current mirrors CM1 and CM2 in every possible sequence.[1] These sequences are DP+CM1+CM2, DP+CM2+CM1, CM1+DP+CM2, CM1+CM2+DP,

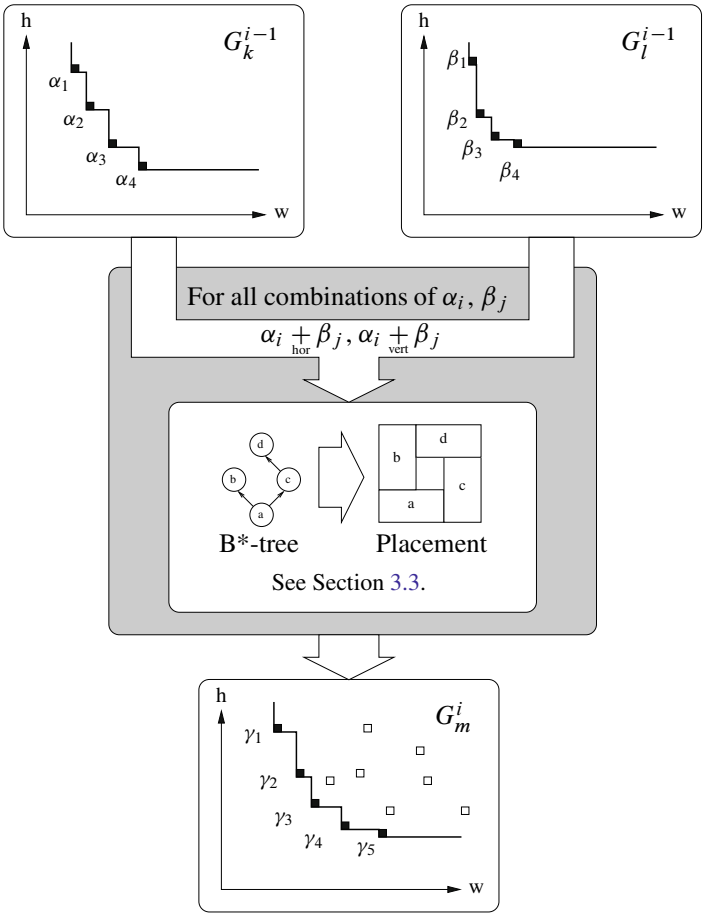


Fig. 3.32 The enhanced shape function addition

[1] It can be shown that enhanced shape function additions are not commutative.

Algorithm 3.4: enumerateOnHierarchyLevelOf(element) ([37], © IEEE 2008)

```
begin
    resultESF ← empty enhanced shape function;
    if element is basic group G_i^0 then
        basicEnumeration(basic group G_i^0);
        store resulting enhanced shape function in resultESF;
    else
        ESFList ← empty list of enhanced shape functions;
        // Generate enhanced shape functions for the children
        forall children of element do
            childESF ← enumerateOnHierarchyLevelOf(child);
            store childESF in ESFList;
        // Try all combinations of the enhanced shape functions
        forall combination sequences of the enhanced shape functions in ESFList do
            forall enhanced shapes in two enhanced shape functions to be added do
                combine B*-trees of the enhanced shapes (Sect. 3.4.3);
                generate placements for B*-trees for evaluation (Sect. 3.3);
                append resulting enhanced shapes to resultESF;
        drop suboptimals from resultESF;
    return resultESF;
end
```

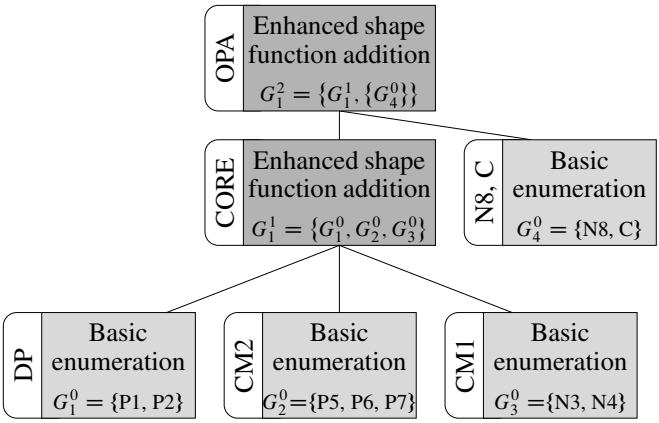


Fig. 3.33 The basic enumerations and enhanced shape function additions for the miller operational amplifier of Fig. 3.30

CM2+DP+CM1, CM2+CM1+DP. Finally, the resulting enhanced shape function is combined in every possible sequence again with the enhanced shape function of the basic group {C, N8} (see Fig. 3.33).

The HSMPG tree determines the order in which the enhanced shape functions are combined. Thus, proximity constraints are fulfilled, because modules will be placed in close proximity, which are close to each other in hierarchy.

Finally, the result of this approach is a set of possible placements for the circuit, having different aspect ratios.

3.6 Experimental Results

The approach proposed in this paper was implemented in C++. All results were computed on a Pentium 4, running at 3.2 GHz with 1,024 MB RAM on Fedora Linux. To demonstrate the approach, placements for different circuits are shown and discussed in Sect. 3.6.1. Publicly available benchmark circuits for analog placement do not yet exist. To compare Plantage with other placement methods, two circuits extracted from [30] are used. First, placements for these circuits are shown and compared to the results of other placement approaches. Then, to demonstrate the effective handling of minimum distance constraints in Plantage, these two circuits are modified by adding wells around symmetry groups. The comparison is discussed in Sect. 3.6.2. Finally, an experiment with piecewise-linear minimum distance constraints is discussed in Sect. 3.6.4.

3.6.1 Discussion of the Presented Approach

To demonstrate the effectiveness of the proposed approach, placements for five different circuits have been generated. The results of the conducted experiments are summarized in Table 3.2. The Examples 3–5 are discussed in more detail describing their constraints and where they can be seen in the placements. The sizings of the modules originate from a large semiconductor manufacturer and are taken from an up-to-date process library. Thus, they can be considered representative for current analog circuits.

Example 1 is a miller amplifier [40], Example 2 is the comparator shown in Fig. 3.10, Example 3 is a folded cascode amplifier [40], Example 4 is a fully differential amplifier similar to the circuit published in [41], and Example 5 is similar to the buffer amplifier published in [42]. The number of devices which form these circuits are shown in Table 3.2. For Example 1, 4, and 5, some devices have been split into subdevices to enable common centroid placement, resulting in additional modules. Furthermore, up to seven different variants are defined for the modules.

First, the HSMPG trees of the circuits have been generated. The number of created groups and the minimum, average and maximum group size is listed in Table 3.2. The HSMPG trees are used by the placement method to split the placement problems into subproblems (see Sect. 3.5). Smaller groups lead to smaller subproblems, which can be solved faster and are therefore preferable. It can be observed that most generated groups are very small, which is advantageous. The SMP graph for circuit 5 contains 14 super nodes after handling the constraint requirements of type M_S to S. The root group, which is created due to proximity constraints from the netlist, is very big and has been further divided by hand for placement.

Placement constraints of the circuit have been generated out of the HSMPG tree. They are listed in Table 3.2. The types cover alignment constraints, which equalize the x- or y-coordinates of two modules, same variant constraints which force two modules to the same variant including same orientation, symmetry constraints

Table 3.2 Summary of example circuits and results generated by the proposed approach

Example	1	2	3	4	5
Name	Miller	Comparator	Folded cascode	Fully differential	Buffer
# of devices	9	10	22	30	42
# of modules	13	10	22	32	46
# of variants per module	3–6	2	2–4	2–3	2–7
Generated HSMPG tree					
# of Groups	5	8	17	19	21
Group size: Min-Max	2–4	2–4	2–3	2–5	2–14
Average	2.6	2.5	2.5	2.2	2.9
Generated constraints					
# Alignment	1	2	8	6	10
# Device proximity	3	5	8	12	14
# Symmetry (Pairs)	2	4	8	10	6
# Common centroid	1	0	0	1	2
# Variant	3	6	11	16	16
# Hierarchical proximity	2	3	9	7	7
Technology constraints					
# Minimum distance	2	1	1	1	1
Computed placements					
# of placements	35	4	12	12	114
Best area usage	115%	110%	121%	129%	111%
Runtimes in seconds					
Constraint generation	0.3	0.3	0.5	0.9	1.2
Plantage	14	1	44	691	134

for pairs of modules, which are given by (3.13), and (3.14) as well as proximity constraints, which make modules stay in close proximity or to form a block in the layout. In addition, minimum distance constraints have been defined between the nMOS and pMOS transistors to reserve space for the n-well.

Finally, placements have been generated using the new methodology. Table 3.2 lists the number of generated Pareto-optimal placements and the achieved area usage. The area usage is the ratio between the area of the bounding rectangle and the area used by modules and wells. Ideally, this value would be 100%. But this is not possible in general due to constraints and device shapes. For the examples, the best achieved result is 111%. The runtimes consumed by the constraint generation and Plantage are shown in the last rows of Table 3.2. It can be seen that the runtime of the constraint generation is small compared to the placer. But even for the largest circuit, having many different module variants and constraints, the placer did not need more than 12 min.

Example 3: Folded Cascode Op Amp

Example 3 is a folded cascode op amp. The schematic and hierarchical placement rules of this circuit and its constraints, together with the HSMPG tree are shown

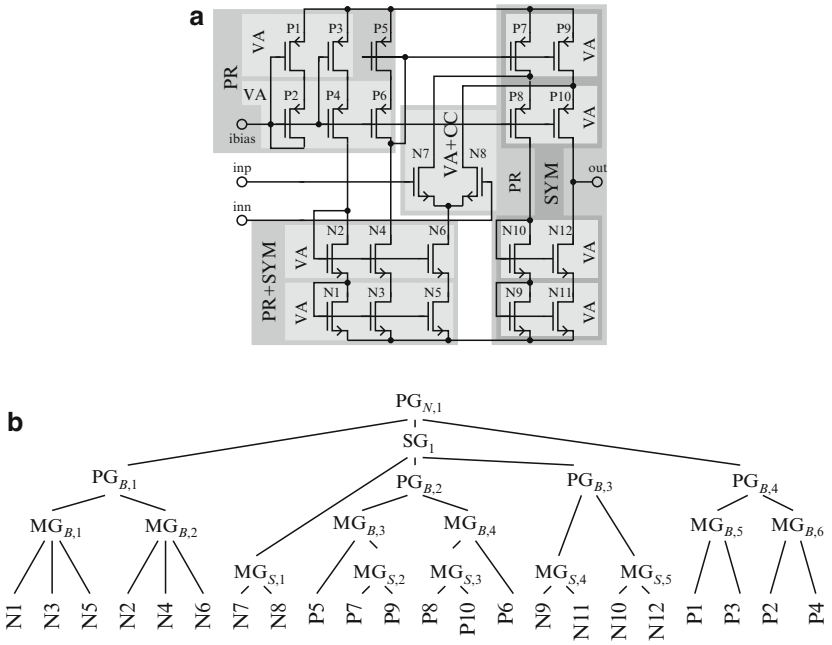


Fig. 3.34 Example 3 ([37], © IEEE 2008): Schematic (**a**) and HSMPG tree (**b**) of the folded Cascode op amp

in Fig. 3.34. The differential pair and the output part (P7–P10 and N7–N12) must be built symmetrically. In the HSMPG tree, this is reflected by $\mathrm{MG}_{S,1}$ to $\mathrm{MG}_{S,5}$, resulting in a symmetry constraint with five symmetrical pairs of transistors as well as variant constraints to achieve matching. For the differential pair, a common centroid constraint is used. Additional variant, alignment, and proximity constraints are defined by the other current mirrors. The generation of the constraints took 0.5 s. Between the nMOS and pMOS transistors, a minimum distance constraint is defined to preserve space for the n well. Plantage generates 12 different placements with different aspect ratios in 44 s. Figure 3.35 shows the shape function and three placements. The symmetry group is colored light gray. The placement is dominated by four big modules P7–P10, causing a corner in the shape function, marked with an "⋆". An example of the close proximity constraints is that N1–N6 must always be placed close to each other. The area usages of the placements are always above 121%, because of empty areas caused by the minimum distance constraints between nMOS and pMOS transistors.

Example 4: Fully Differential Amplifier

Example 4 (Fig. 3.36a) is a fully differential amplifier similar to the one published in [41]. It is characterized by a high degree of symmetry [43]. In the generated

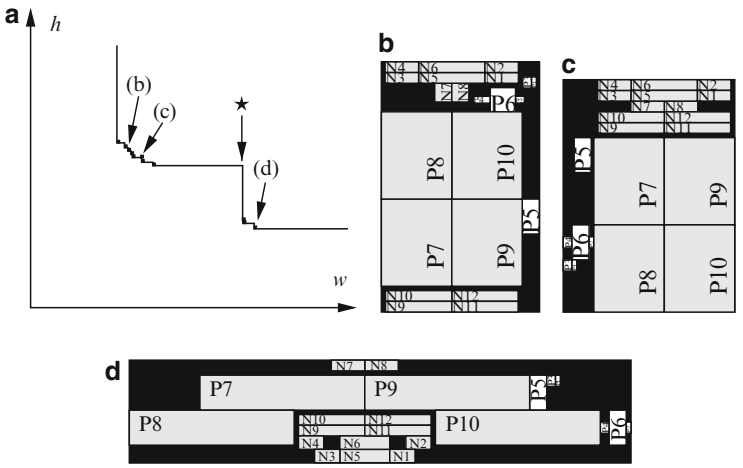


Fig. 3.35 Example 3 ([37], © IEEE 2008): placements (**b**)–(**d**) and the corresponding shape function (**a**)

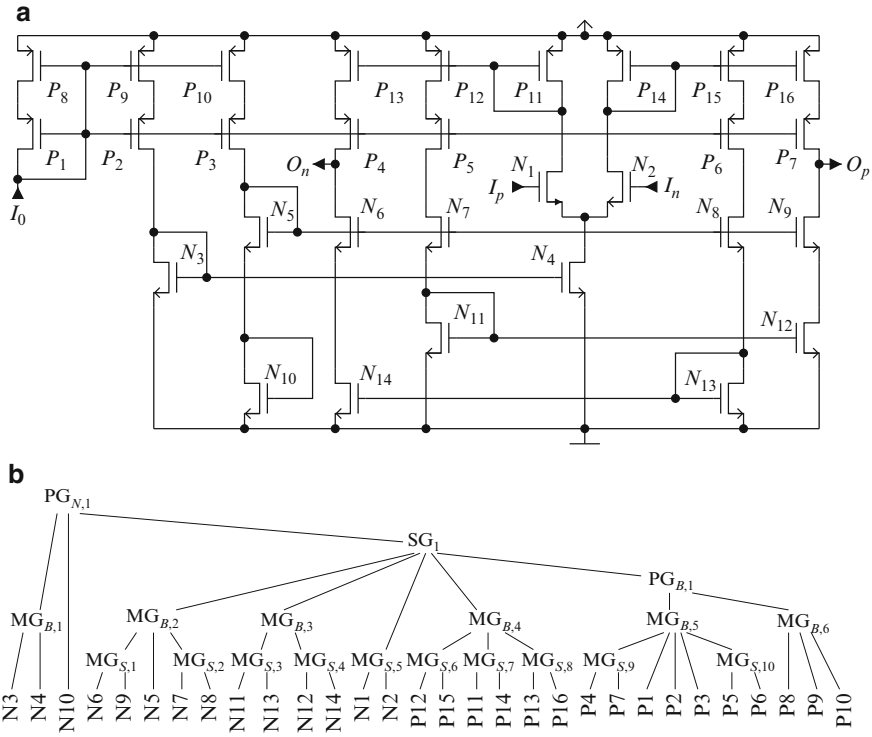

a
b
P_1 P_2 P_3 P_4 P_5 P_6 P_7 P_8 P_9 P_{10} P_{11} P_{12} P_{13} P_{14} P_{15} P_{16}
N_1 N_2 N_3 N_4 N_5 N_6 N_7 N_8 N_9 N_{10} N_{11} N_{12} N_{13} N_{14}
I_0 I_p I_n O_n O_p
$PG_{N,1}$ SG_1 $PG_{B,1}$
$MG_{B,1}$ $MG_{B,2}$ $MG_{B,3}$ $MG_{B,4}$ $MG_{B,5}$ $MG_{B,6}$
$MG_{S,1}$ $MG_{S,2}$ $MG_{S,3}$ $MG_{S,4}$ $MG_{S,5}$ $MG_{S,6}$ $MG_{S,7}$ $MG_{S,8}$ $MG_{S,9}$ $MG_{S,10}$
N3 N4 N10 N6 N9 N5 N7 N8 N11 N13 N12 N14 N1 N2 P12 P15 P11 P14 P13 P16 P4 P7 P1 P2 P3 P5 P6 P8 P9 P10

Fig. 3.36 Example 4: Schematic (**a**) and HSMPG tree (**b**) of the fully differential Opamp

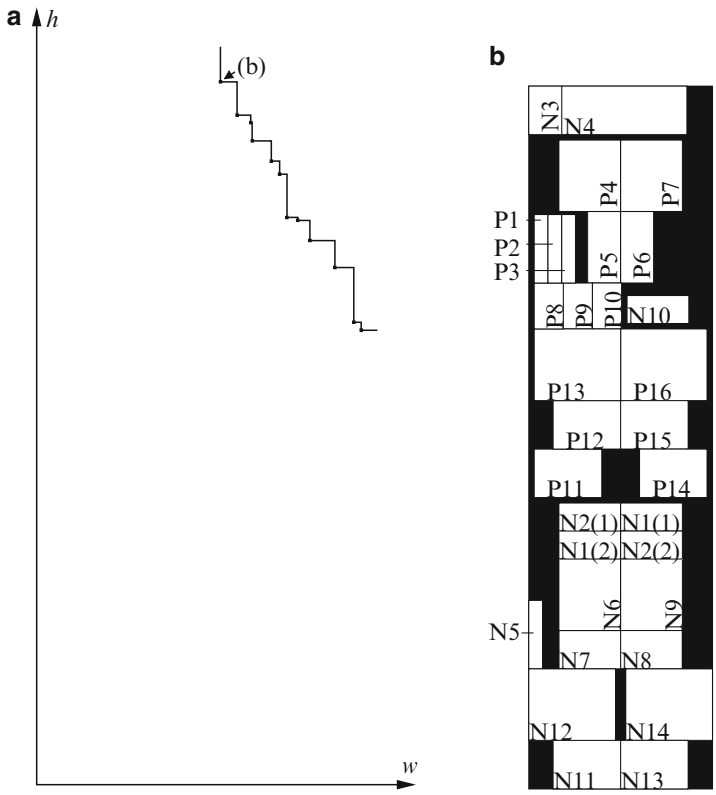


Fig. 3.37 Example 4: Shape function (**a**) and placement (**b**)

HSMPG tree (Fig. 3.36b), this is reflected by the symmetry group SG_1 and the matching groups (symmetry) $MG_{S,1}$ to $MG_{S,10}$. The building blocks of the circuit, mainly simple current mirrors and level shifters, determine matching groups $MG_{B,1}$ to $MG_{B,6}$. In addition to the generated constraints, a minimum distance constraint between nMOS and pMOS transistors was defined. Figure 3.37 shows some placements of the amplifier together with the shape function. All placements are quite high and narrow because of the symmetry groupSG_1.

Example 5: Buffer Amplifier

Example 5 is a CMOS buffer amplifier similar to [42], shown in Fig. 3.38. Figure 3.38b shows the generated hierarchical placement rules. Before starting the automatic placement, the top proximity group $PG_{N,1}$ has been manually divided into smaller groups. The two differential pairs DP1 (N1, N2) and DP2 (P3, P4) are

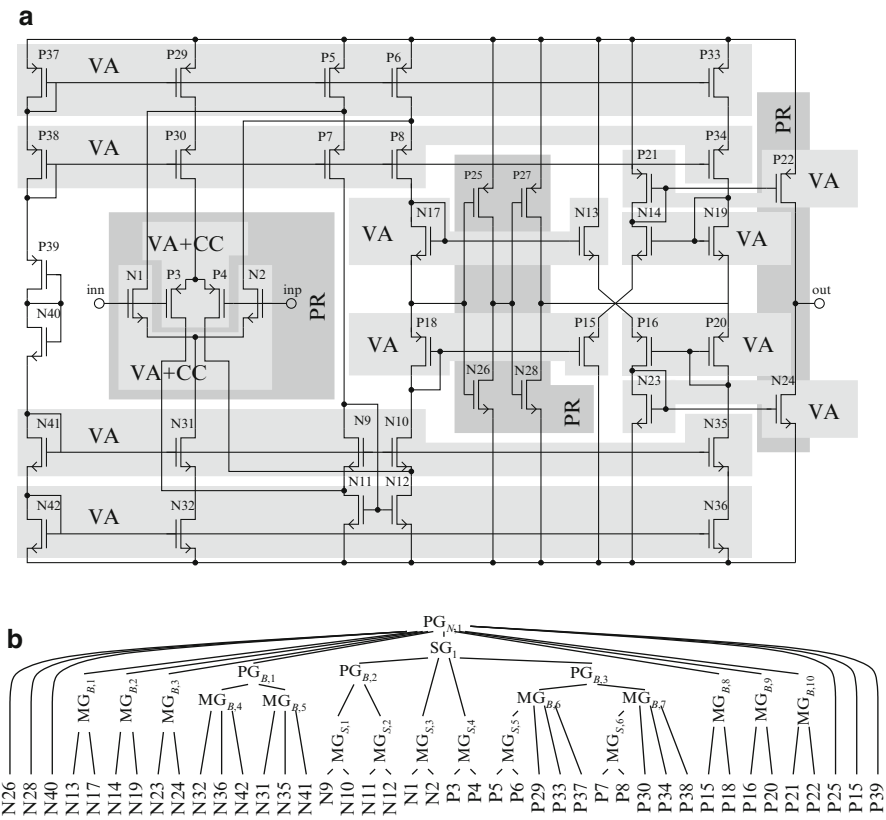


Fig. 3.38 Example 5 ([37], © IEEE 2008): Schematic (**a**) and hierarchical placement rules (**b**) of the buffer amplifier similar to [42]

realized by two common centroid arrays (N1a, N1b, N2a, N2b) and (P3a, P3b, P4a, P4b), respectively. Three placements are shown in Fig. 3.39b–d. As demonstrated by the figures, a minimum distance constraint is kept between nMOS and pMOS transistors to reserve area for the wells. The differential pairs DP1 and DP2 are colored light gray in the placements. DP1 and DP2 are placed in close proximity, since they are close to each other in hierarchy. The same applies to the modules P22 and N24, as well as P25, N26, P27, N28, marked in dark gray. For several transistors, different variants with different sizings are given representing different numbers of fingers of the transistor gate. For example, P39 has three different aspect ratios in the shown placements. Figure 3.39a shows the shape function for this circuit. It represents 114 different placements, having different aspect ratios. Plantage calculates these placements in 134 s. The HSMPG tree was built in 1.2 s.

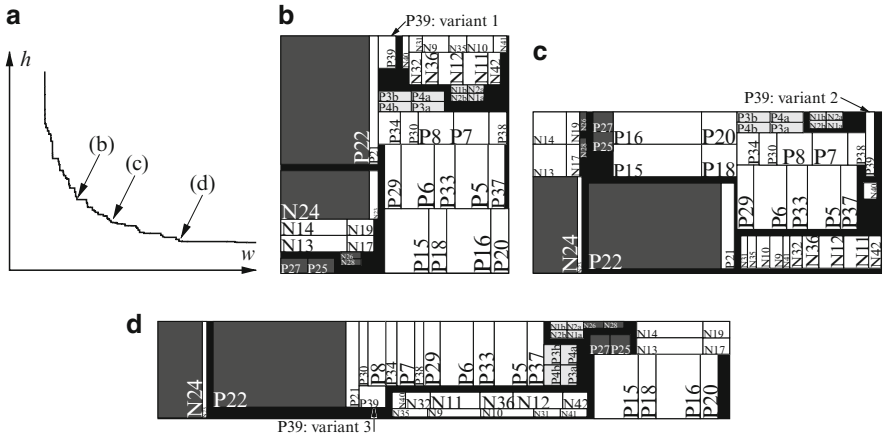


Fig. 3.39 Example 5 ([37], © IEEE 2008): placements (**b**)–(**d**) and the corresponding shape function (**a**)

3.6.2 Comparison with Other Approaches

This approach is compared with other approaches in the following. For that reason, placements were generated using Plantage for the circuits named "biasynth_2p4g" and "lnamixbias_2p4g" used in [22, 23, 30, 32, 33, 44–46]. The module sizings were extracted from [22]. Plantage generates ten placements in 337 s for "biasynth_2p4g". For "lnamixbias_2p4g", 32 placements are generated in 387 seconds. Since no netlist information is given, no HSMPG tree can be calculated. Thus, balanced trees with four to six children per node are used as hierarchy trees. The placements with the lowest area usages are shown in Figs. 3.40 and 3.41, respectively. The symmetry groups within these circuits are colored in different shades of gray. The runtimes and area usages of other approaches are taken from [45] and [32]. For the given examples, the total module area is constant. In this case, the area usage is inversely proportional to the total area, and is taken as a quality metric.

The results are summarized in Tables 3.3 and 3.4. It can be seen from the numbers that the area usages of this approach are approximately equal to the area usages of the best recently published placers. The area usage of Plantage is 1% and 2% higher than the best for the two examples, respectively, but approximately 10% lower than area usage of other approaches. It should be noted that Plantage is the only deterministic approach among the approaches used for comparison.

It is difficult to compare the runtimes of this approach with the runtimes of other approaches, because they were measured on different computers. Since other approaches use simulated annealing, their runtimes may vary from run to run for different seeds, but no mean values and standard deviations are known for the runtimes of other approaches. In contrast, the runtimes of Plantage are constant for each example. On all accounts, the runtimes of Plantage, even for the big circuit

Fig. 3.40 Result of "biasynth_2p4g" obtained by Plantage ([37], © IEEE 2008) (time: 5.6 min, area usage: 104.96%)

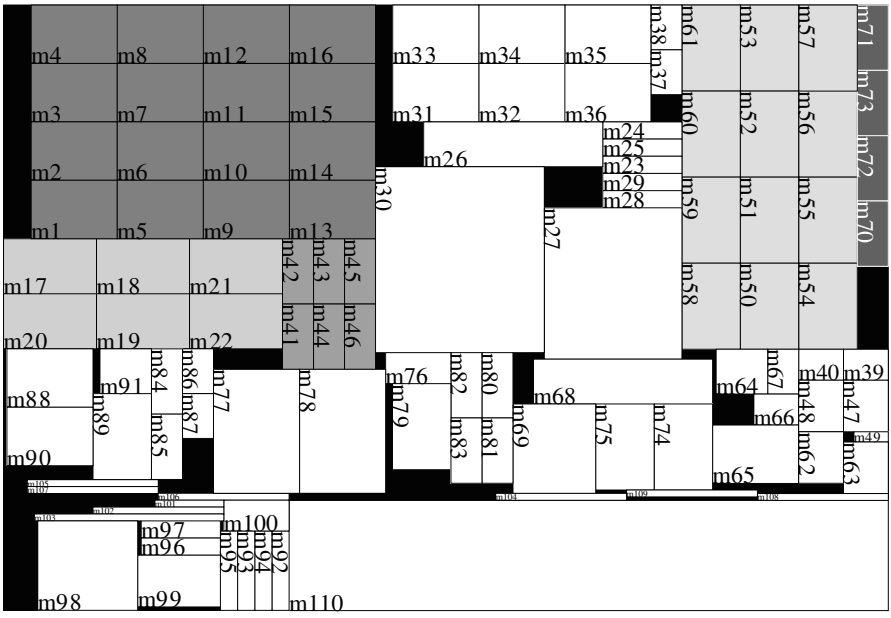


Fig. 3.41 Result of "lnamixbias_2p4g" obtained by Plantage ([37], © IEEE 2008) (rotated by 90° clockwise) (time: 6.4 min, area usage: 107.68%)

"lnamixbias_2p4g" (110 modules), allow interactive use. In addition, the time to set up a new design is short because the constraints and the HSMPG tree are generated automatically based on the netlist. Therefore, Plantage is fit for industrial application.

Table 3.3 Description of the two example circuits [37].

Circuit description			
Name	No. of modules	No. of sym. mods.	Mod. area (10^3 μm^2)
biasynth_2p4g	65	8 + 12 + 5	4.70 $\widehat{=}$ 100%
lnamixbias_2p4g	110	16 + 6 + 6 + 12 + 4	46.00 $\widehat{=}$ 100%

Table 3.4 Comparison of area usage and runtimes for different approaches, based on two industrial circuits ([37], © IEEE 2008)

Results of different approaches				
	biasynth_2p4g		lnamixbias_2p4g	
Approach	Area	Time	Area	Time
Sequence pair [30]	114.89	780*	110.43	2,824*
Segment tree with segment tree [22]	114.89	246*	109.35	726*
Sequence pair and linear programming [44]	106.38	403†	108.59	3,252†
Sequence pair with dummy nodes [33]	118.51	134†	113.50	227†
Symmetry islands [45]	104.68	22†	105.72	43†
Sequence pair with Johnson's priority queue [32]	N/A	N/A*	109	480*
Plantage	*104.96*	*337*†	*107.68*	*387*†

In [32], only the placement of "lnamixbias_2p4g" is shown. No area usage is given in that paper. For comparison, the area usage in Table 3.4 was calculated based on Fig. 3 of [32]. All times are measured in seconds (times which are marked with *star* were measured on a Sun Blade 100, 500 MHz, and times which are marked with *dagger* were measured on a Pentium 4, 3.2 GHz) and all area usages in % of the total module area.

3.6.3 Experiment with Linear Minimum Distance Constraints

To demonstrate the effective handling of minimum distance constraints in Plantage, "biasynth_2p4g" and "lnamixbias_2p4g" are modified: In the modified circuits, each symmetry group is located in a separate well. Thus, various minimum distance constraints need to be considered between the modules. The resulting placements are shown in Figs. 3.42 and 3.43. For example, Module m1 in Fig. 3.43 must keep a distance of d_{well} to all modules outside of its own well, and a distance of $2 \cdot d_{\text{well}}$ to all modules being located in other wells.

Considering the minimum distances for these wells results in higher computational effort. Thus, the generation of placements takes 76% more CPU time (593s) for "biasynth_2p4g", and 72% more CPU time (664 s) for "lnamixbias_2p4g". From a practical point of view, the resulting placements are still compact. With the CPU times still being in the range of minutes even for these large circuits, industrial application is possible. Since other approaches do not address minimum distance constraints in detail, no comparisons can be given.

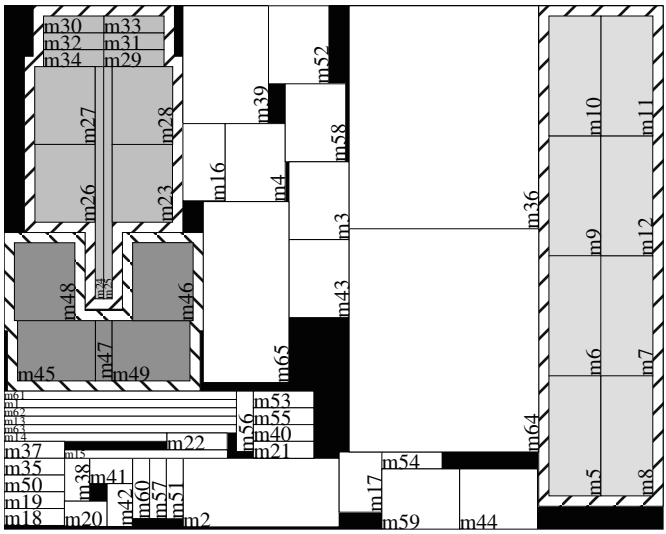


Fig. 3.42 Result of modified "biasynth_2p4g" with wells (time: 9.9 min, area usage: 107.74%)

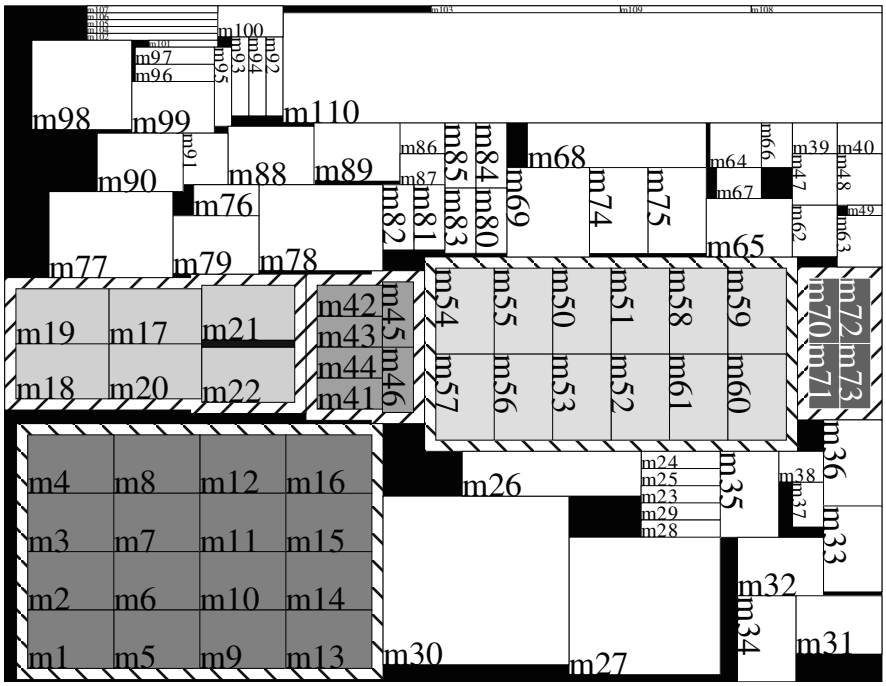


Fig. 3.43 Result of "lnamixbias_2p4g" with wells (time: 11.1 min, area usage: 109.24%)

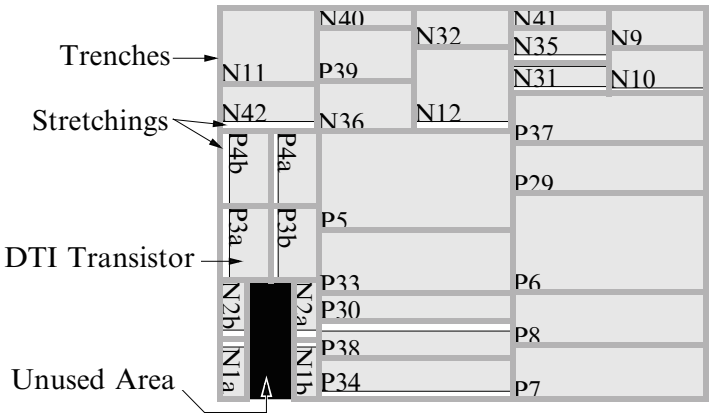


Fig. 3.44 DTI transistor example similar to [42]

3.6.4 Experiment with PWL Minimum Distance Constraints

An example consisting of DTI transistors is shown in Fig. 3.44. The modules are shown as gray rectangles. White areas are caused by the stretching of the DTIs. The circuit placement shows an unused area in the lower left part of the layout. The circuit is similar to the input stage of the buffer amplifier of [42], consisting of 30 modules. The placement was generated in approximately 15 min, because the linear MIP solver is significantly slower than the Simplex solver. The area usage of this circuit is 110%. Although the runtime increases within reasonable bounds when considering PWL minimum distance constraints, this methodology still produces compact placements.

3.7 Conclusion

In this chapter, Plantage, a new deterministic approach for analog circuit placement including a new method to generate placement constraints was introduced. A HSMPG tree is presented, which represents a circuit as hierarchical groups and the placement constraints inside each group. The generation process starts by identifying basic building blocks and generating symmetry conditions. These results are used to model the constraints of the circuit in a graph of requirements with respect to symmetry, matching, and proximity (SMP graph). This graph is used together with an importance order of the constraints to generate a hierarchical tree of symmetry, matching, and proximity groups (HSMPG tree).

The hierarchy is used to guide a bottom-up enumeration efficiently. All placements are enumerated for small parts of the circuit. New concepts, the enhanced shape functions, and the enhanced shape additions are used to combine these

placements efficiently with a recursive algorithm based on the hierarchy. In contrast to other approaches, the final result of Plantage is a set of placements with different aspect ratios instead of a single solution. An algorithm is presented in this chapter, which generates a placement for a B*-tree considering linear as well as piecewise-linear constraints.

Plantage considers device-proximity, symmetry, common centroid, minimum distance, and variant constraints. This approach is the first to handle all these constraints deterministically. The results of this approach show an area usage, which is comparable to the best-published results of other placers. Plantage generates results in reasonable time allowing industrial application.

References

1. John M. Cohn, David J. Garrod, Rob A. Rutenbar, and L. Richard Carley. *Analog Device-Level Layout Automation*. Kluwer, Dordecht, 1994.
2. Alan Hastings. *The Art of Analog Layout*. Prentice-Hall, Englewood Cliffs, NJ, 2001.
3. Enrico Malavasi and Alberto L. Sangiovanni-Vincentelli. Area Routing for Analog Layout. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(8):1186–1197, August 1993.
4. L. Clavelier, B. Charlet, B. Giffard, and M. Roy. Deep trench isolation for 600 V SOI power devices. In *Conference on European Solid-State Device Research*, pages 497–500, September 2003.
5. E. Charbon, E. Malavasi, and A. Sangiovanni-Vincentelli. Generalized constraint generation for analog circuit design. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 408–414, 1993.
6. U. Choudhury and A. Sangiovanni-Vincentelli. Automatic generation of parasitic constraints for performance-constrained physical design of analog circuits. In *IEEE/ACM International Conference on Computer-Aided Design and Manufacture of Electronic Components*, pages 208–224, February 1993.
7. Enrico Malavasi, Edoardo Charbon, Eric Felt, and Alberto L. Sangiovanni-Vincentelli. Automation of IC Layout with Analog Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):923–942, August 1996.
8. D. J. Chen and B. J. Sheu. Generalised approach to automatic custom layout of analogue ICs. In *Circuits, Devices and Systems, IEE Proceedings G*, volume 139, pages 481–490, August 1992.
9. Qinsheng Hao, Sheqin Dong, Song Chen, Xianlong Hong, Yi Su, and Zhiyi Qu. Constraints generation for analog circuits layout. *2004 International Conference on Communications, Circuits and Systems*, 2:1339–1343, volume 2, June 2004.
10. Bogdan G. Arsintescu. A Method for Analog Circuits Visualization. In *IEEE International Conference on Computer Design (ICCD)*, pages 454–459, 1996.
11. M. E. Kole, J. Smit, and O. E. Herrmann. Modeling symmetry in analog electronic circuits. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 315–318, May 1994.
12. Su Yi, Sheqin Dong, Qingsheng Hao, Xiangqing He, and Xianlong Hong. Automated Analog Circuits Symmetrical Layout Constraint Extraction by Partition. In *ASIC, 2003. Proceedings. 5th International Conference on*, volume 1 of *1*, pages 166–169, October 2003.
13. Tobias Massier, Helmut Graeb, and Ulf Schlichtmann. The Sizing Rules Method for CMOS and Bipolar Analog Integrated Circuit Synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(12):2209–2222, December 2008.
14. D. W. Jepsen and C. D. Gellat Jr. Macro Placement by Monte Carlo Annealing. In *IEEE International Conference on Computer Design (ICCD)*, pages 495–498, 1983.

15. John M. Cohn, David J. Garrod, Rob A. Rutenbar, and L. Richard Carley. KOAN/ANAGRAM II: New Tools for Device-Level Analog Placement and Routing. *IEEE Journal of Solid-State Circuits SC*, 26(3):330–342, March 1991.
16. Koen Lampaert, Georges Gielen, and Willy M. Sansen. A Performance-Driven Placement Tool for Analog Integrated Circuits. *IEEE Journal of Solid-State Circuits SC*, 30(7):773–780, July 1995.
17. Enrico Malavasi, Edoardo Charbon, Eric Felt, and Alberto L. Sangiovanni-Vincentelli. Automation of IC Layout with Analog Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):923–942, August 1996.
18. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.
19. Florin Balasa. Modeling Non-Slicing Floorplans with Binary Trees. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 13–17, University of Illinois at Chicago, Dept. of EECS, November 2000.
20. Pei-Ning Guo, Chung-Kuan Cheng, and Takeshi Yoshimura. An O-Tree Representation of Non-Slicing Floorplan and Its Applications. In *ACM/IEEE Design Automation Conference (DAC)*, volume 36, pages 268–273, June 1999.
21. Yun-Chih Chang, Yao-Wen Chang, Guang-Ming Wu, and Shu-Wei Wu. B*-Trees: A New Representation for Non-Slicing Floorplans. In *ACM/IEEE Design Automation Conference (DAC)*, volume 37, pages 458–463, 2000.
22. Florin Balasa, Sarat C. Maruvada, and Karthik Krishnamoorthy. On the Exploration of the Solution Space in Analog Placement With Symmetry Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):177–191, February 2004.
23. Po-Hung Lin, Yao-Wen Chang, and Shyh-Chang Lin. Analog Placement Based on Symmetry-Island Formulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(6):791–804, June 2009.
24. H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. VLSI Module Placement Based on Rectangle-Packing by the Sequence-Pair. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(12):1518–1524, 1996.
25. S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajitani. Module Placement on BSG-Structure and IC Layout Applications. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 484–493, 1996.
26. X. Hong, G. Huang, Y. Cai, J. Gu, S. Dong, C.-K. Cheng, and J. Gu. Corner Block List: An Effective and Efficient Topological Representation of Non-Slicing Floorplan. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, November 2000.
27. Qiang Ma, Evangeline F. Y. Yong, and K. P. Pun. Analog Placement with Common Centroid Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, November 2007.
28. Jai-Ming Lin and Yao-Wen Chang. TCG-S: Orthogonal Coupling of P-admissible Representations for General Floorplans. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(6):968–980, June 2004.
29. Yingxin Pang, Florin Balasa, Koen Lampaert, and Chung-Kuan Cheng. Block Placement with Symmetry Constraints based on the O-tree Non-Slicing Representation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 464–468, June 2000.
30. Florin Balasa and Koen Lampaert. Symmetry Within the Sequence-Pair Representation in the Context of Placement for Analog Design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(7):721–731, July 2000.
31. Karthik Krishnamoorthy, Sarat C. Maruvada, and Florin Balasa. Fast Evaluation of Symmetric-Feasible Sequence-Pairs for Analog Topological Placement. In *5th IEEE Int. Conf. on ASIC (ASICON)*, pages 71–74, 2003.
32. Karthik Krishnamoorthy, Sarat C. Maruvada, and Florin Balasa. Topological Placement with Multiple Symmetry Groups of Devices for Analog Layout Design. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2032–2035, May 2007.

33. Yiu-Cheong Tam, Evangeline F. Y. Young, and Chris Chu. Analog Placement with Symmetry and Other Placement Constraints. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, November 2006.
34. David A. Johns and Ken Martin. *Analog Integrated Circuit Design*. John Wiley & Sons, 1997.
35. Frank Harary. *Graph Theory*. Addison-Wesley series in mathematics, 1969.
36. Brian S. Everitt. *Cluster Analysis*. Edward Arnold, 3 edition, 1993.
37. Martin Strasser, Michael Eick, Helmut Graeb, Ulf Schlichtmann, and Frank M. Johannes. Deterministic Analog Circuit Placement using Hierarchically Bounded Enumeration and Enhanced Shape Functions. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 306–313, November 2008.
38. R. H. J. M. Otten. Efficient Floorplan Optimization. In *IEEE International Conference on Computer Design (ICCD)*, pages 499–501, October 1983.
39. Gerhard Zimmermann. A New Area and Shape Function Estimation Technique for VLSI Layouts. In *ACM/IEEE Design Automation Conference (DAC)*, volume 25, pages 60–65, 1988.
40. Kenneth R. Laker and Willy Sansen. *Design of Analog Integrated Circuits and Systems*. McGraw-Hill, New York, 1994.
41. Ivano Galdi, Edoardo Bonizzoni, Piero MALCOVATI, Gabriele Manganaro, and Franco Maloberti. 40 MHz IF 1 MHz Bandwidth Two-Path Bandpass $\Sigma\Delta$ Modulator With 72 dB DR Consuming 16 mW. *IEEE Journal of Solid-State Circuits SC*, 43(7):1648–1656, July 2008.
42. J. Fisher and R. Koch. A Highly Linear CMOS Buffer Amplifier. *IEEE Journal of Solid-State Circuits SC*, 22:330–334, 1987.
43. Michael Eick, Martin Strasser, Helmut Graeb, and Ulf Schlichtmann. Automatic Generation of Hierarchical Placement Rules for Analog Integrated Circuits. In *ACM/SIGDA International Symposium on Physical Design (ISPD)*, March 2010.
44. Shinichi Kouda, Chikaaki Kodama, and Kunihiro Fujiyoshi. Improved Method of Cell Placement with Symmetry Constraints for Analog IC Layout Design. In *ACM/SIGDA International Symposium on Physical Design (ISPD)*, April 2006.
45. Po-Hung Lin and Shyh-Chang Lin. Analog Placement Based on Novel Symmetry-Island Formulation. In *ACM/IEEE Design Automation Conference (DAC)*, pages 465–470, June 2007.
46. Po-Hung Lin and Shyh-Chang Lin. Analog placement based on hierarchical module clustering. In *ACM/IEEE Design Automation Conference (DAC)*, pages 50–55, June 2008.

Part II
Routing

Chapter 4
Routing Analog Circuits

Günhan Dündar and Ahmet Unutulmaz

Abstract This chapter presents a review of routers for analog circuits, some practical issues for analog routing, and a template-based routing strategy. Basic algorithms and methods used for routing are discussed first, starting from the maze router and continuing towards more sophisticated routing algorithms. Then, data representations commonly used for routing are described in some detail.

Analog design specific routing issues and methods are then discussed. Various routing strategies from the literature and developed by the authors are presented in some detail. Specialized routers for two analog applications, namely RF design and analog arrays, are also presented. Manufacturing and yield issues for routing are discussed briefly before conclusions and a discussion of various open problems in routing of analog integrated circuits.

4.1 Introduction

Routing is one of the final steps in analog layout synthesis. Its main objective is to electrically connect terminals of the layout modules—transistors, capacitors, differential pairs, etc.—and input/output ports. Due to the fact that the performances of an analog circuit are critically dependent on layout parasitics, the routing of an analog circuit requires more attention than that of a digital circuit. On the other hand, since routing is one of the final steps, the quality of the routing and the final performance of a routed layout is strongly affected by all the preceding synthesis steps.

Layout of the differential input stage of an OPAMP is shown in Fig. 4.1a, a router is supposed to add the wires filled with black for this layout. Device merging reduces the required connections and also the parasitics. If the transistors are merged, number of the required wires decreases as displayed in Fig. 4.1b. Even for

G. Dündar (✉)
Department of Electrical and Electronic Engineering, Boğaziçi University, Bebek 34342, Istanbul, Turkey
e-mail: dundar@boun.edu.tr

H.E. Graeb (ed.), *Analog Layout Synthesis: A Survey of Topological Approaches*,
DOI 10.1007/978-1-4419-6932-3_4, © Springer Science+Business Media, LLC 2011

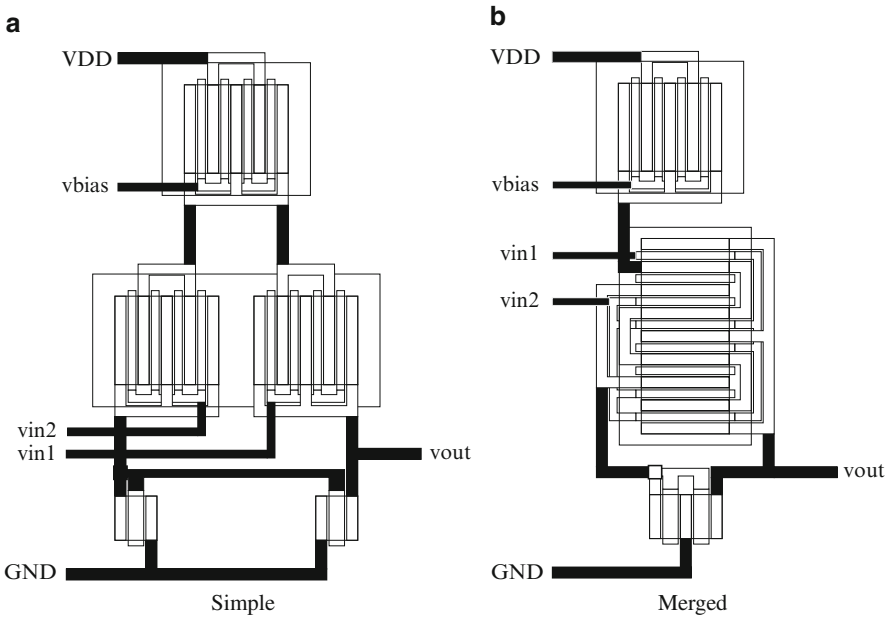


Fig. 4.1 Quality of synthesized layout strictly depends on the synthesis steps before routing, such as: folding, merging, placement, and shaping of the devices. Routing overhead depicted as black rectangles in (**a**) decreased considerably in (**b**) after merging some of the transistors

this simple analog circuit, it is evident that the quality of the routing highly depends on folding, merging, placement, and shapes of the devices. Thus, routing must be integrated with device generation and placement steps as much as possible. Although one may use a very sophisticated routing algorithm for the layout in Fig. 4.1a, its performance is bound to be worse than that of Fig. 4.1b. In addition to a review of analog routers, some practical issues for analog routing will be presented in this chapter. Basic algorithms and methods used for routing are discussed in Sect. 4.2, and Sect. 4.3 describes some representations required for routing. Routing issues and methods for analog circuits are mentioned in Sect. 4.4, and Sect. 4.5 contains a review of the routing strategies. Specialized routers—for Analog Arrays and RF circuits—are briefly mentioned in Sect. 4.6. Manufacturing and yield issues for routing are discussed in Sect. 4.7. Finally, Sect. 4.8 concludes this chapter.

4.2 Basic Routing Algorithms

The routing problem in layout design involves the process of formally defining the precise conductor path necessary to properly interconnect the associated nets of the system. This section concerns itself with basic routing algorithms utilized in various applications. The routing algorithms used in analog integrated circuits are in essence the same as those in digital integrated circuits, or even printed circuit

boards. However, analog integrated circuits typically contain fewer paths, but more constraints on each path. Thus, these basic algorithms are tailored accordingly to obtain routes for analog integrated circuits.

4.2.1 Maze Router

One of the earliest and most well-known algorithms for wire routing is Lee's maze routing algorithm [1]. Actually, this algorithm is an extension of the earlier work by Moore [2] to uniform grids. The maze router starts by numbering every grid point starting from the source. The wave consisting of increasing numbers is propagated until the wavefront reaches the target. Then, the route is obtained by backtracing the numbers. In Fig. 4.2, a wave starting from source S is propagated to the target T and by backtracking the path filled with gray is found. In this figure, the assigned numbers and letters (X) denote the wave numbers and the obstacles, respectively.

Lee's algorithm is guaranteed to find a solution to the routing between two nets if such a solution exists. In addition, this solution will be the shortest path. However, its application to large grids requires extremely large memory structures as well as having a time complexity of $O(nm)$, where n and m are the dimensions of the grid. Here, each grid point must have a location in memory representing the layer(s) assigned to that location. In spite of these drawbacks, Lee's algorithm has been the dominant solution in wiring, both at the PCB level and at the IC level for a long time. In the meanwhile, researchers have suggested techniques to increase the efficiency of the algorithm, in terms of both speed and memory usage over the years [3–7].

				5	6	7	8	9	10	11	12	13	14					
			3	4	5	6	7	8	9	10	11	12	13	14				
		3	2	3	4	X	X	X	10	11	12	13	14	15	16			
	3	2	1	2	3	4	5	X	11	12	13	14	15	16	17	18		
3	2	1	S	1	2	3	4	X	X	X	X	X	X	17	18	19	20	
4	3	2	1	2	3	4	5	X	23	22	21	20	19	18	19	20	21	
5	4	3	2	3	4	5	6	X	24	23	22	21	20	19	20	21	22	
6	5	4	3	4	5	6	7	X	23	22	T	20	21	20	21	22	23	
7	6	5	4	5	6	7	8	X	22	21	20	19	20	21	22	23	24	
8	7	6	5	6	7	8	9	X	21	20	19	18	19	20	21	22	23	
9	8	7	6	7	8	9	10	X	X	X	X	17	18	19	20	21	22	
10	9	8	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	
11	10	9	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	
	11	10	9	10	11	12	13	14	15	16	17	18	19	20	21	22		
		11	10	11	12	13	14	15	16	17	18	19	20	21	22			
			11	12	13	14	15	16	17	18	19	20	21	22				
				13	14	15	16	17	18	19	20	21	22					

Fig. 4.2 A wave starting from source S is propagated to the target T. X and the assigned numbers denote the obstacles and the wave numbers, respectively. By backtracking, the path filled with gray is found

4.2.2 Line Expansion Routers

One major improvement over the basic idea was to represent the search space in terms of line segments rather than grid points. By this method, generally called line-search, both run time and memory space requirements can be reduced drastically since a memory location is not allocated to each grid point any more. Furthermore, line searching algorithms tend to reduce the number of unnecessary bends in the routes; however, they do not guarantee the shortest paths. The first algorithms were suggested independently by [8] and [9]. Unfortunately, line searching algorithms become more and more cumbersome as the routing becomes more complicated and the memory and computation time requirements may even surpass those of the maze router. Furthermore, [9] does not guarantee to find a solution even if one exists. In Fig. 4.3, the general methodology of line searching algorithms is described. Here, horizontal and vertical line segments are added around the source (S) and target (T) ports and the obstacles. Then, these line segments are combined to construct a path from S to T. An excellent discussion of maze routing as well as a review of the above algorithms can be found in [10].

4.2.3 Channel Routing

A channel router is a specific router for integrated circuits and it is commonly used in digital routing. Although it does not directly consider the layout parasitics, early analog routers have applied this approach. As the name implies, the routing is done in channels, where a channel is a horizontal area with fixed pins on the top and bottom. The channel height is not specified, but calculated by the router. Numbers

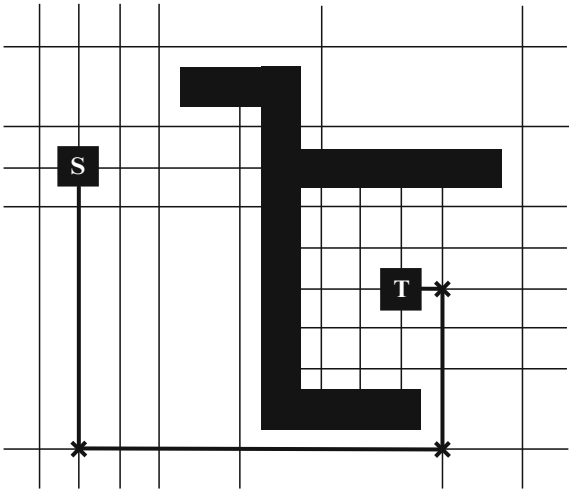


Fig. 4.3 Horizontal and vertical line segments are added around the obstacles and the ports. These line segments are combined to construct a path

are assigned to the pins and these numbers indicate the nets corresponding to these pins. Thus, those pins with different numbers must be electrically isolated to prevent short circuits between different nets.

Given the positions and nets of the pins, two constraint graphs can be extracted, namely: vertical constraint graph (VCG) and horizontal constraint graph (HCG). VCG handles the vertical positioning of nets in a channel and it has a vertex v_x for each net x. A directed edge $e = (v_x, v_y)$ is added between two vertices, if two pins in net x and net y overlap horizontally and the pin in net x is above the pin in net y. On the other hand, HCG handles the horizontal spans of the nets and unlike VCG, it is an undirected graph. Vertices in HCG correspond to the nets, the edges are undirected. An edge $e = (v_x, v_y)$ is added between vertices v_x and v_y if horizontal span of net x overlaps with that of net y. The VCG and HCG may be used to completely represent an instance of the channel routing problem. Horizontal span of the nets for the channel routing problem in Fig. 4.4a is shown in Fig. 4.4b, also VCG and HCG are shown in Fig. 4.4c and 4.4d, respectively. If VCG does not contain any edges, the routing may be fully done by the Left-Edge Algorithm [11]. One of the solutions to the channel routing problem in Fig. 4.4a is depicted in 4.4e; this solution is obtained by applying the Constrained Left-Edge Algorithm, a modified version of the Left-Edge Algorithm. This algorithm can only handle noncyclic constraints in the VCG. If there are cyclic constraints, the problem becomes more complex and there are several approaches to solve it, such as [12–14]. Although these approaches are used for routing of digital circuits, they are not commonly preferred in analog tools, due to poor estimation of parasitic effects.

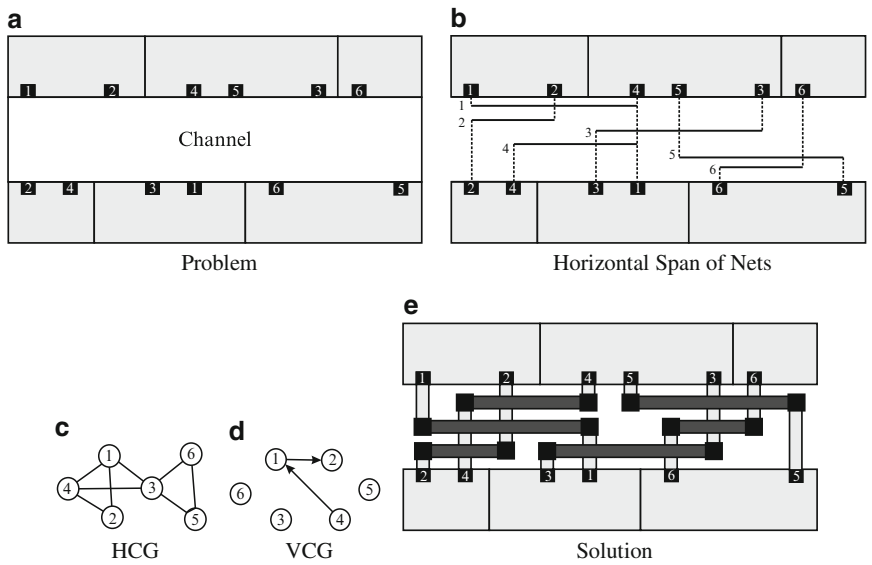


Fig. 4.4 Channel routing: (**a**) an instance of channel routing problem (**b**) horizontal span of the nets (**c**) horizontal constrint graph (HCG) (**d**) vertical constraint graph (VCG) (**e**) A solution to the problem in (**a**)

4.2.4 Steiner Routing

The maze and line expansion routing algorithms are developed to connect two nets to each other in the presence of obstacles. Thus, to complete the routing of a case with many nets to be connected in pairs, the algorithm should be applied sequentially. However, the quality, or even existence of a solution depends on the order on which the pairs are selected. This problem is called net ordering and is discussed in the ensuing sections. Furthermore, routing nets with more than two terminals optimally is an NP-complete problem, usually called the Steiner Problem, named after the Swiss mathematician Jacob Steiner (1796–1863), who solved the problem of joining three villages by a system of roads having minimum total length [15]. Although Jacob Steiner's solution was an independent work, the same problem had already been attacked and solved by several earlier mathematicians. The corresponding solution called the Steiner-minimal-tree (SMT) is the tree connecting the nets with minimum wirelength. Steiner-minimal-tree problem is simplified to minimum-rectilinear-Steiner-tree problem (MRST), when only horizontal and vertical paths are allowed. Compared to SMT, MRST is more efficient in terms of computational costs. The MRST solution should not be confused with a rectilinear minimum spanning tree (MST), where all wires are from terminal to terminal, with no intermediate junctions. SMT, MRST, and the rectilinear-MST are depicted in Fig. 4.5. It is shown in [16] that in the worst case rectilinear-MST solution is 1.5 times longer than the MRST solution. The SMT problem has many applications, thus it is a well-studied problem about which hundreds of papers and several books have been written. Hence, many heuristics to this NP-complete problem exist. One would be to find an approximate Steiner point between these nets and to apply maze routing between the terminals and the Steiner point. Another approach would be initially applying a maze router between a pair of points. Once a path is found between these points, this path can be used as a new source for the wavefront. Finally, all the above discussion has assumed that a single layer was available for routing and that only horizontal and vertical wires are allowed. One of the first extensions to this routing strategy was diagonal routing proposed in [17]. Over the years, multiple routing layers were introduced, performance criteria started to be taken into account, and algorithms tailored specifically to integrated circuits

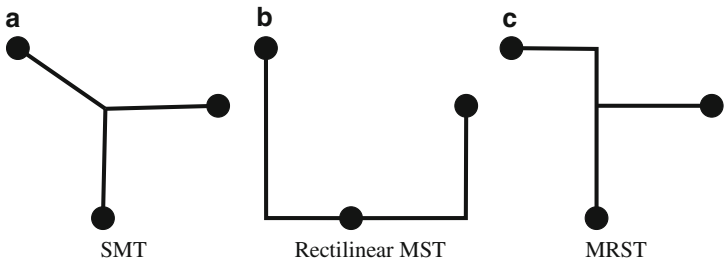


Fig. 4.5 Comparison of trees: (**a**) SMT, (**b**) Rectilinear MST, (**c**) RST

with special emphasis on various technology issues were developed. The interest in routing has not disappeared and there is still much room for improvement as seen by a recent global routing competition in [18].

4.3 Representations

The layout of an integrated circuit is quite a complex entity, and much attention must be paid to the data representations utilized for any operation. It is well known that the performance and/or complexity of an algorithm depend on how the data are represented. This section initially discusses techniques for representing the layout itself and compares these techniques. Then, the issue of representing the connectivity information is treated. Finally, the representation of layout rules is briefly discussed.

4.3.1 Layout Representations

There are traditionally three representations of layout for the routing problem, which are grid-based, tile-based, or topological representations. These representations are described in the following paragraphs.

4.3.1.1 Grid-Based Representations

Grid-Based layout representation is very simple and suitable for the area routing problem; thus, it is very common in the literature. This representation is mainly classified into two subgroups, namely, uniform and nonuniform grid representations. The uniform representation has been used since the first routers – based on Lee's Maze Algorithm – and the nonuniform representation has been used since 1980s.

Uniform Grid Representation

The simplest grid-based representation is on a uniform grid. However, routing on a uniform grid typically makes the wiring unnecessarily area hungry as well as using too much memory. Using a fine grid will result in dense layouts, but requires excessive amount of memory, whereas using a coarse grid will result in larger layouts, but the memory requirements are less. As shown in Fig. 4.6, the coarse grid is not sufficient to represent the layouts of some wires, having horizontal and vertical dimensions comparable to the size of the grid, precisely. On the other hand, the fine grid consumes huge amount of memory.

Moreover, variable width routing and layout constraint observance are necessary for ensuring circuit performance. This is not the case for digital layouts,

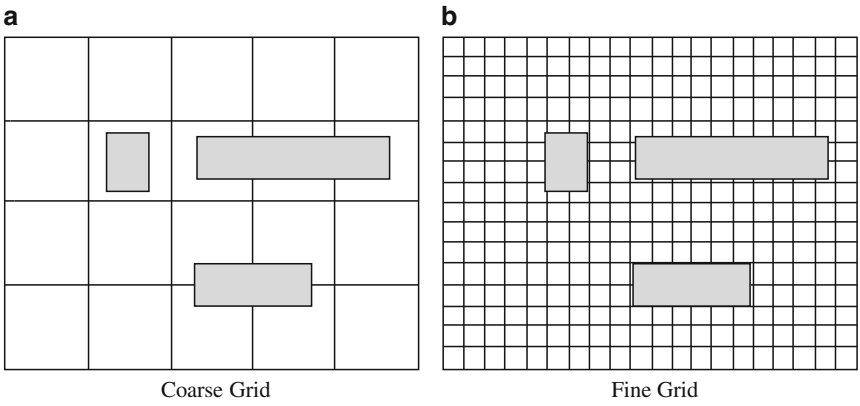


Fig. 4.6 Uniform grids may be (**a**) coarse or (**b**) fine

where block sizes, wiring widths, and terminal locations can be fixed without considerable performance loss. Several improvements on the fixed-width grid have been suggested among which are routing on a uniform grid and compaction to obtain a grid-free layout. However, these small improvements cannot yield satisfactory layouts.

Nonuniform Grid Representation

A major class of grid-based representations are nonuniformly sized grids. The variable size grid is more efficient than the uniform one in terms of memory usage. In addition to its efficiency, the size of the grid may be much finer than a uniform grid in the congested areas. Moreover, compared to the tile-based structures, it yields faster evaluation of the parasitic effects. This is due to the fact that the neighborhood information is embedded in the regular structure.

The variable size grid is constructed by extending the boundaries of the layout components to the left, right, bottom, and top boundaries of the layout as shown in Fig. 4.7a. The number of the grid cells may be reduced if the component boundaries are used as blocks for the boundary extension process, such as done in [19] and depicted in Fig. 4.7b.

Graph Theory is one of the basic concepts in computational science; thus, there are a variety of algorithms in this subject. Due to this fact, graphs are commonly used in Electronic Design Automation (EDA). Making use of these graph theory-based algorithms, for the routing problem, is only possible if the constructed grid is converted to a graph. Grid Graph $G(V, E)$ is consctucted (Fig. 4.7c), where V is the set of *empty cell*s and E is the set of edges connecting *neigboring cell*s. Any partition of a grid that cannot be further partitioned is called a *cell*. An empty cell is defined as any cell in the grid that does not contain any layout component. All the blank cells in Fig. 4.7b are empty cells. Moreover, two cells are neigbours

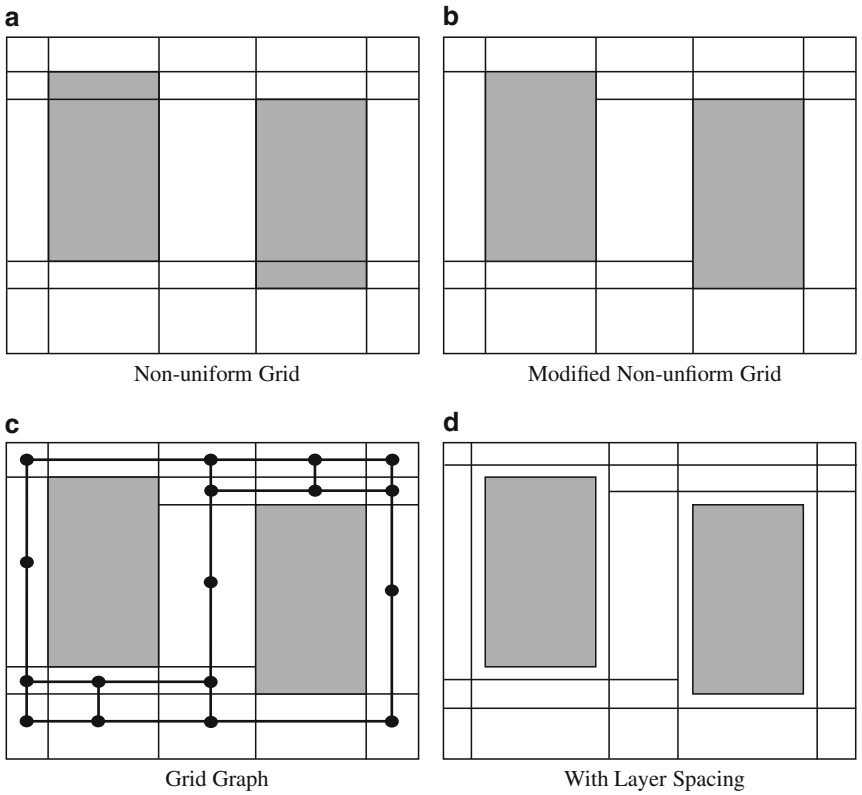


Fig. 4.7 Nonuniform grid is constructed by extending the boundaries of the layout components to the *left*, *right*, *bottom*, and *top* (**a**) number of grids may be reduced as in (**b**) A grid graph extracted from the grid in (**a**) is depicted in (**c**). In (**d**) some space around the components is left empty to prevent design rule violations

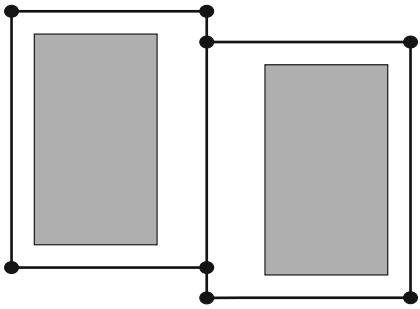


Fig. 4.8 Channel intersection graph is based on the channels around the layout components and it is suitable for line expansion-based routing. Vertices are the intersection point of these channels, and the edges are the channel segments between the vertices

when they have overlapping boundaries. When the graph $G(V, E)$ is constructed it is used to find a path between the cells in V that correspounds to a route in the layout. A variant of the Grid Graph is the Channel Intersection Graph (Fig. 4.8). In this graph, vertices are the intersection point of the channels around the layout

components, and the edges are the channel segments between these vertices. This type of graph represents only the channels between layout components and it is used with line expansion-based routing algorithms.

The described grid generation process ignores the design rules, where design rule considerations are left to the path-finding algorithm. However, a simple extension on the grid may be used to add the minimum space constraints into the Grid Graph. As shown in Fig. 4.7d, dimensions of the layout components are updated by adding the minimum space requirements for the layer of the component. Note that the dimensions of the source and target terminals, being routed, are not updated. Otherwise the space, added around these terminals, will not allow the router to connect them.

4.3.1.2 Tile Based Representations

Tile-based representations extend the concept of a grid by allowing different granularities for the blocks. This representation is based on the corner stiching data structure [20], which ties the layout components by adding pointers at their corners. Similar to the grid-based representation, this representation stores empty spaces as well as layout components. The tile-based representation suits line expansion algorithms better, such as in ANAGRAM II [21].

The main advantage of this representation is that it helps in expediting layout synthesis as blocks can have very different dimensions; for example, capacitors are much larger than other blocks. In such a case, a grid model will be inefficient, and the memory ocupation will be much higher. Similar to the grid graph of the grid representation, a tile graph $G(V, E)$ is extracted to be used with the graph algorithms. Figures 4.9a, b depict a sample representation and the corresponding tile graph, respectively. Although this representation is memory efficient, it is more complicated. Also, it may be required to add *escape points*[1] during the process of path searching, which slightly enlarges the tile graph of the representation [22]. Effects of the escape points to the path length is shown in Fig. 4.9. A path before adding the escape points is shown in Fig. 4.9c and a path after adding the escape points is shown Fig. 4.9d.

4.3.1.3 Topological Representations

In the previous sections, it was suggested that grid-based and tile-based notations have difficulties in expressing constraints on the relative positions of blocks. Topological representations, including slicing trees [23], the sequence-pair (SP) descriptions [24], and O-trees [25], present the relative positioning of blocks. Thus, they are very efficient in modeling constraints on block placements including symmetries. Although these representations are used in placement, some of them are also proposed for simultaneous placement and routing, such as, SP, multi-layer sequence-pair (Multi-SP) [26], channeled-bounded sliceline grid (Channeled-BSG)

[1] If a point is not a corner of a tile, it is called an escape point.

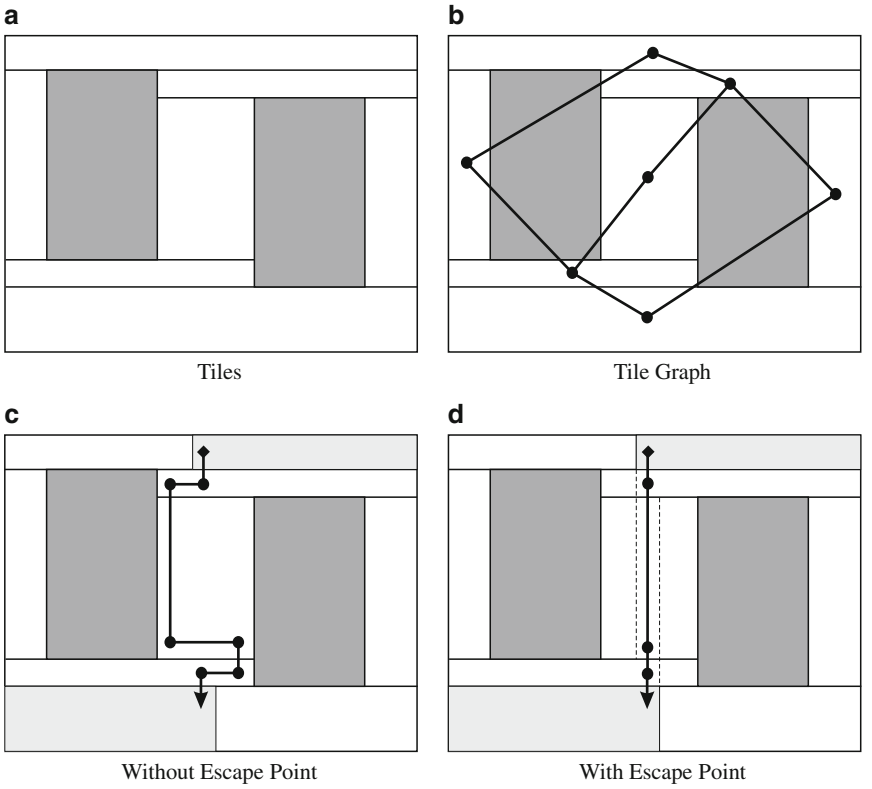


Fig. 4.9 Tiles for a layout are depicted in (**a**) the corresponding tile graph is in (**b**) Effects of escape points are indicated in (**c**) and (**d**)

[27]. However, these representations are poor in extracting layout parasitics because block and wire vicinities are only partially expressed in these notations. Thus, topological representations seem to be more suited to placement than routing.

4.3.2 Connectivity Representation

Routers often remove previous routes and add new ones – *rip-up and re-route*– to improve the quality of the routing, which changes the *connectivity information*[2] frequently. Moreover, in every iteration of the router, checking the connectivity through layout extraction will not be efficient in terms of time complexity. Thus, it is required to handle the connectivity information in a special data structure.

[2] Connectivity information refers to the physical connectivity information. Being in the same net does not imply connectivity.

The basic requirements for such a data structure may be listed as follows:

- Net-splitting will allow routing the high current paths seperately, which prevents voltage drops on critical wires.
- Servicing of connectivity information should be fast.
- The nearest connectable layout component to a given component must be found efficiently.

The netlist for a circuit is represented with set $C = \{N_{Gnd}, N_{Vdd}, N_i, \ldots\}$, where the N_i's denote the nets. Every net (N_i) in C is a set of subnets[3] S_{ij} as $N_i = \{S_i\ , S_{i2}, S_{ij}, ...\}$. S_{ij} are composed of groups (G's). Formulation of the S_{ij} is $S_{ij} = \{G_{ij1}, G_{ij2}, G_{ijk}, \ldots\}$. The groups G_{ijk} contains the layout components (e_l's). A schematic diagram for the nets of a circuit, C, is shown in Fig. 4.10. The G sets are used in handling the connectivity information, such that being in the same group means physical connection has been done. However, being in the same subnet but in different groups indicates an expectation for a physical connection. The layout components e_1 and e_2 shown in Fig. 4.10b are in the same group G_{111}. This indicates a physical connection between them. However, the components e_1 and e_3 in the same figure are not in the same group but they are in the same subnet S_{11}.

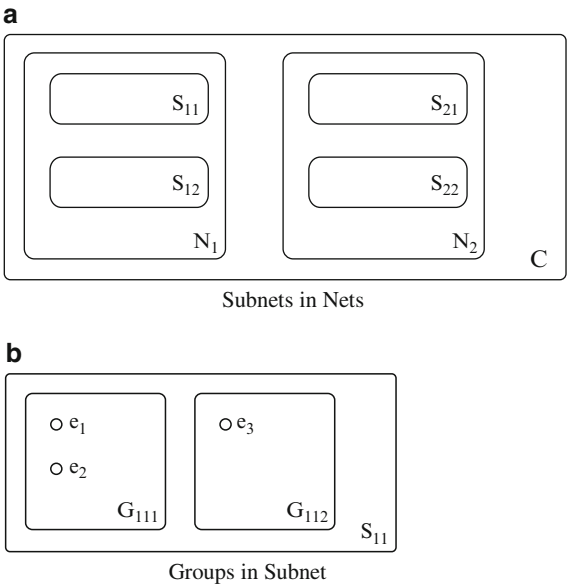


Fig. 4.10 Schematic diagram of the structure: (**a**) netlist is composed of nets, and nets are composed of subnets, (**b**) each subnet contains groups. If two layout components are in the same group, they are physically connected

[3] A subnet is defined to achieve net-splitting and it covers the wires of a net on which the current densities are similar.

This condition indicates these components are going to be connected, but they are not connected yet. Using the defined structure, checking the connectivity information is very efficient, whereby it is done by controlling whether the components are in the same group. Similarly, the nearest unconnected component is found through a search in a given subnet.

4.3.3 Rule Representations

Most analog routers consider parasitic effects, such as wire resistances and crosstalk capacitances. In the extraction of these parasitics, the electrical characteristics of the layout components are frequently checked. Also during the construction of the layout representation and path finding, design rules are frequently needed. Thus, the time complexity of the router is directly related to the service time of the design rules and electrical rules. For the data structure shown in Fig. 4.11, the layer identifiers are used as keys, this data structure returns the requested rule in constant time. This structure is a hierarchy of *Hash Tables*,[4] where the Hash Table in the

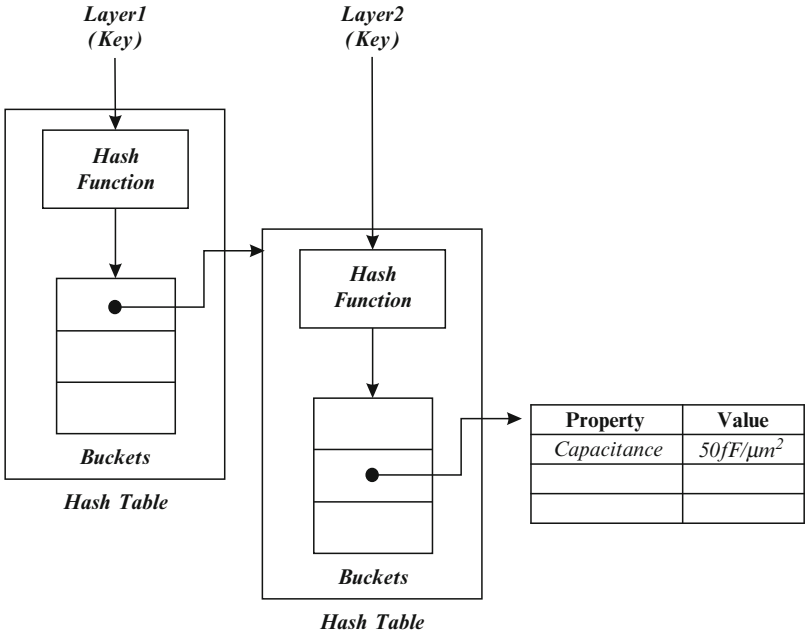


Fig. 4.11 Data structure for rules

[4] A Hash Table is a data structure, which uses a hash function to efficiently map keys to associated values. The hash function is used to transform the key into the index of an array element.

first level stores the addresses of the Hash Tables of the second level that store the rules. The structure shown in Fig. 4.11 is equivalent to the pseudo code of `get_capacitance_between(layer1, layer2)`, which returns the capacitance between *layer1* and *layer2*.

4.4 Routing Issues and Techniques for Analog Circuits

Performances of analog circuits are extremely sensitive to layout parasitics, which are undesired effects due to physical properties of elements. Some of the parasitic effects introduced due to routing and techniques for reducing these effects are discussed in [19, 21, 28–31]. Although it is impossible to eliminate the existence of routing parasitics, it is possible to reduce their effects with the proposed techniques. Splitting paths that are carrying high currents from the ones carrying low currents or routing nets symmetrically are such techniques. In this sub-section, well-known routing techniques and technology limitations are going to be described.

4.4.1 Net Splitting

Current densities in different portions of analog circuits may vary excessively. Thus, there may be an order of magnitude difference between the current densities of two paths in the same net. The resistance of a wire may be modeled as

$$R_{\text{wire}} = \frac{\Delta l * R_{sh}}{\Delta w}, \tag{4.1}$$

where Δl is the length and Δw is the width of the wire and R_{sh} is the sheet resistance. If this model is used for the grounded wires in Fig. 4.12a, there will be a voltage drop of $V_{\text{drop}} = R_{\text{wire}} * I$ on the wire between the dashed lines. Because of the voltage on the wire, the potential at the top end of the wire will be different from the ground potential. This potential difference may vary with time and affect the operation of the analog circuit. Therefore, it must be minimized. One approach [28] is to separate the paths as in Fig. 4.12b and a different approach is to use wider wires for the high current paths as in Fig. 4.12c.

4.4.2 Symmetric Routing

Analog circuit designers frequently introduce topological symmetries in differential circuits to optimize offset, differential gain, and noise. As the analog layout tools have progressed, routers with the capability of symmetric routing were

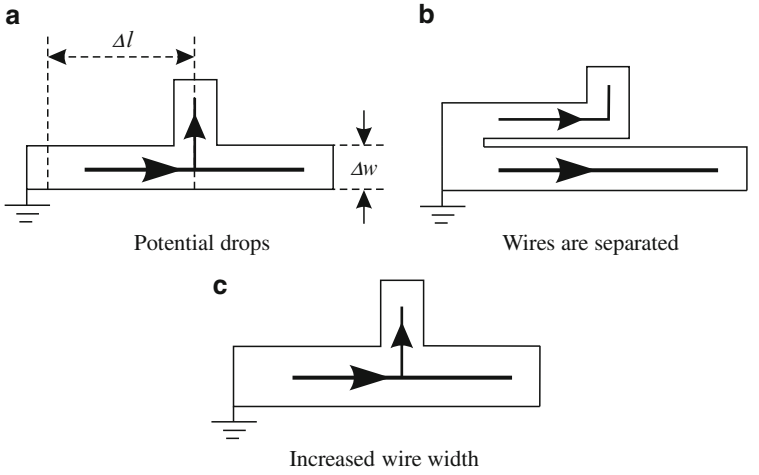


Fig. 4.12 Current densities of wires in a net (**a**) branching wire is effected due to potential drop in the wire segment between the *dashed lines* (**b**) branching wire is separated from the line on which high currents flow (**c**) wire widths are increased to reduce the potential drop on the wire

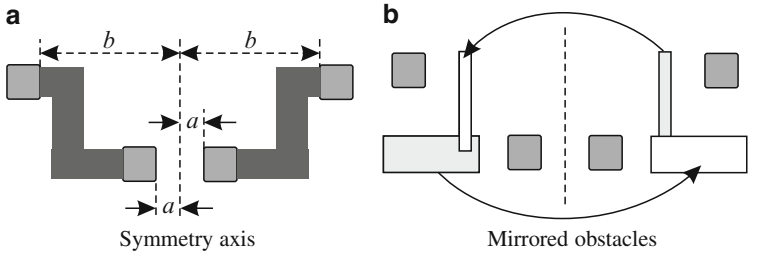


Fig. 4.13 Symmetry axis and nonsymmetric obstacles (**a**) the terminals are symmetric with respect to the *dashed* symmetry axis (**b**) nonsymmetric obstacles are mirrored

developed [30]. These initial tools were followed by tools, such as ANAGRAM II [21] and ROAD [19], having the capability of routing symmetrically in the presence of nonsymmetric obstacles.

Symmetric routing may be considered if only the terminals of the paths being routed are symmetric with respect to a symmetry axis. Figure 4.13a shows a symmetric layout, where the terminals are in gray, and the symmetry axis is the dashed line. This axis splits the layout into two halves.

Initially, presence of nonsymmetric (with respect to symmetry axis) obstacles were not considered [30]. In the absence of nonsymmetric obstacles, the tools route one half of the layout and then mirror the new paths to the other half of the layout.

Later approaches [19, 21] considered nonsymmetric obstacles as well. The approach of ANAGRAM II [21] is to evolve both halves of a symmetric path simultaneously. With this extension, the line-expansion algorithm used by ANAGRAM II became slightly more complex. Line expansion algorithm requires checking

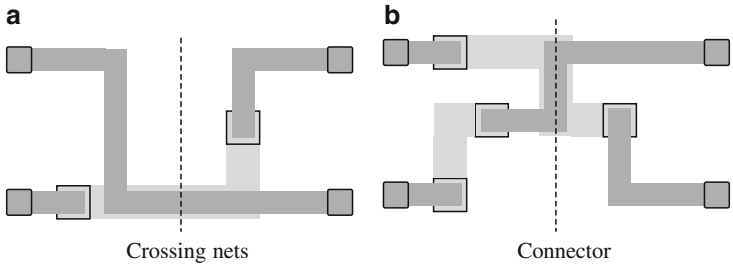


Fig. 4.14 Matching parasitics of crossing-symmetric wires (**a**) matching is poor between symmetric nets (**b**) matching is much better if a "connector" is used

whether both halves of the symmetric path can be legally placed on each side. In this approach, a blockage that exists only on one side of the symmetry line effectively becomes a blockage for the other side as well. Another approach used in ROAD [19] is mirroring the obstacles (Fig. 4.13b) and then routing only for one side. Then, the constructed path is mirrored to the other half.

An acceptable symmetric routing will not be possible with the mentioned approaches if the terminals are on different sides of the symmetry axis and the paths cross. In Fig. 4.14a, such a case is demonstrated. Fortunately, good parasitic matching can be obtained between the nets with the technique described in [31]. A *connector* (Fig. 4.14b) allows two symmetric segments to cross over the axis. Although resistances and capacitances of the two nets are matched, there may be some difference between the parasitics of the symmetric nets due to capacitive and inductive coupling with the other nets. However, this structure is much better than the one depicted in Fig. 4.14a.

4.4.3 Crosstalk and Shielding

Crosstalk between the nets may severely degrade the performance of an analog circuit; thus, it is required to extract these effects during the routing. There are 1-D, 2-D, 2.5-D, and 3-D extraction methods for parasitic capacitances [32]. The 1-D extraction simply uses the equation

$$C_{1D} = A * C_{\alpha} + S * C_{\beta}, \tag{4.2}$$

where A is the area of the overlapping region between two wires, S is the perimeter of this region, C_{α} is the capacitance per unit area, and C_{β} is the fringing capacitance per unit length. The overlapping area is the dashed rectangle in Fig. 4.15. The 2-D extraction also includes capacitances due to nonoverlapping wires. If the 2-D model is used for extraction, total capacitance for the first vertical wire in Fig. 4.15 is given as

$$C_{2D} = C_{1D} + \frac{\Delta l * C_{\gamma}}{d}, \tag{4.3}$$

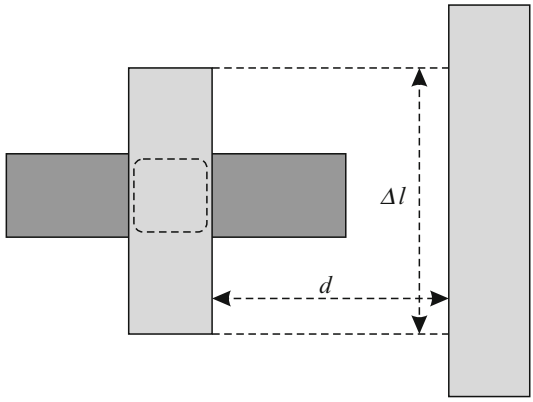


Fig. 4.15 2-D extraction involves overlaping and nonoverlaping capacitances

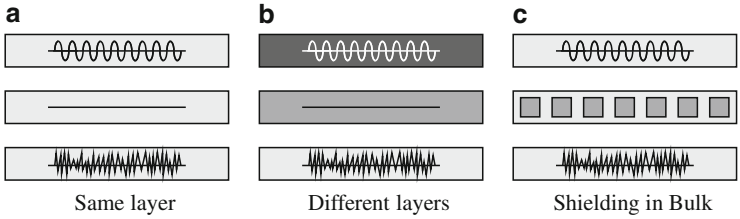


Fig. 4.16 Shielding reduce crosstalk (**a**) shielding on the same layer (**b**) shielding between different layers (**c**) shielding to avoid crosstalk through bulk. *Squares* are contacts to bulk

where C_γ is the crosstalk capacitance per unit length, Δl is the length of overlap in the vertical axis, and d is the separation between the wires. This model is commonly used in routing, due to it is simplicity.

In 2.5-D extraction, fringing effects are considered in advance through the cross-sections of the real 3-D structure. On the other hand, a library—including parameterized 3-D geometric structures—is constructed in 3-D extraction and extracted geometries from the layout are matched with the ones in the library. Although 3-D extraction is more accurate than the mentioned extraction methods, it may be cumbersome for path searching in routing due to its time complexity.

In RF circuits, inductive coupling may also be critical for the performance and RLC models for interconnects, such as in [33] are used to observe these inductive effects. Even EM simulations may be carried out to observe the parasitic effects more accurately.

As discussed in the preceding paragraphs, long wires running in parallel affect the performance of the analog circuits. One way to reduce the crosstalk between these wires is to add space between them. If it is impossible to reduce the coupling through adding space, the router may introduce a shield between the critically coupled nets. The router ROAD [19] has the capability to add a shield line between wires (Fig. 4.16a). In Fig. 4.16, three different shielding methods are displayed. These methods may be used to reduce the crosstalk via the bulk or routing layers. Note that, not to worsen the effects of crosstalk, the shielding wire must be connected to a dc potential or it must be grounded.

4.5 Routing Strategies for Analog Circuits

This section covers various strategies for routing analog circuits. The discussion starts from digital-like routers and proceeds to more complex optimization-based strategies, roughly following a historical flow.

4.5.1 Digitally Inspired Early Routing Strategies

Advances in computer performance and algorithm theory as well as the need to design complex circuits in a short time led to the appearance of the first design automation tools for analog circuits in the late 1980s. Most of these tools also included automated layout tools as a part of the tool suite. Automated routing tools were also incorporated into the layout tools. The available experience at that time led designers to develop digitally inspired routing strategies. The main characteristics of these routers are a two-step routing approach and heuristic cost functions used in a mostly feedforward manner.

Routing in digital circuits is typically performed in a two-step strategy; global routing and detailed routing. A single step maze routing approach was generally not preferred in the early analog routers because it is computationally less efficient than channel routing. Moreover, since there is no global routing phase, nets are considered individually, and there is no global view of the interconnection problem. Since a channel router routes all the nets at the same time, one achieves routing of higher quality, compared to other routing strategies that route one net at a time. While using channel routing for complete layouts, the problem is broken into smaller problems of routing individual channels, resulting in much faster routing compared to the general area-routing algorithms. Channel routing also allows for changes in placement of blocks with relative ease during detailed routing.

This early two-step approach is evident in the ROSA router of the LADIES automatic layout system [34], where routing areas are extracted and decomposed into rectangles, initially. An adjacency graph is then formed from these rectangles. Global routing is achieved by finding the shortest path on the adjacency graph. Finally, the detailed routing is performed using the river routing algorithm. Here, analog constraints are not seriously taken into account during the routing.

The constraints imposed on acceptable solutions generally involve one or more of the following: (i) total area of interconnection, (ii) amount of crosstalk, (iii) number of crossovers and vias, and (iv) density of wiring. In these early studies, the driving factor of routing, as in placement, was the minimization of crosstalk. To achieve this end, the tools tried to route the sensitive and noisy nets separately as well as trying to minimize the crossover. Also, designer knowledge was somehow incorporated into most routing tools, whereby the designer had to identify various nodes or interact with the tool in the actual placement of the paths. In summary, the approach taken was to augment digital routing methods with some additional constraints, which take signal-coupling reduction into account. Nevertheless, the evaluation of the routing

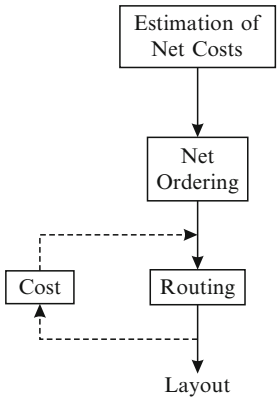


Fig. 4.17 Estimation of the costs was used in a feedforward approach. In some cases, the costs are used in a weak feedback loop

parasitics and alternatives in this class of tools do not go far beyond a qualitative evaluation of the nets. In addition to the rather sketchy estimation of the costs, they were used in a feedforward approach, (See Fig. 4.17) or in some cases in a weak feedback loop. The feedback loop is called weak because it addresses intermediate variables, such as overlap areas, which interact heuristically with the final performance rather than the actual performance parameters. Thus, the performance of the layout within a certain specification region was not guaranteed.

On the other hand, a priori estimation of the costs, however they may be primitive, has a major advantage in that the net ordering problem can be addressed. As mentioned in previous sections, the order in which nets are routed has a major impact on the overall routing solution. Furthermore, routing more critical nets initially allows them to occupy more privileged locations in the layout, thus minimizing their length and possible crosstalk. These routes then act as obstacles to later and less critical nets to be routed, whose routing inevitably becomes less efficient.

Such a net ordering approach was presented in MIGHTY [35], which is a part of the OPASYN [36] tool. The order that the nets are routed is determined a priori by classifying the nets into several categories according to their functions, such as input nets, output nets, etc. MIGHTY uses a rip-up and reroute strategy for routing; that is, when a particular net pair meets congestion, previously routed nets are ripped up and rerouted to make space. The authors see this approach to be feasible due to the relatively few nets in analog design.

A contemporary of OPASYN is IDAC, which has its companion layout tool called ILAC [37, 38]. In ILAC, nets are classified into four categories; namely, sensitive nets, noisy nets, noncritical signal nets, and power supply nets. A net ordering strategy is also employed in ILAC. Power nets are routed first, followed by sensitive nets, and then noncritical nets. Noisy nets are routed last, while the power nets and noncritical nets provide shielding between the critical and the noisy nets. The global router in ILAC is a maze router. However, it handles net couplings, undesired crossings, planarity (for power nets), and congestion. After global routing is completed, channel sizes are estimated depending on the number of nets to be routed inside the channel. The detailed router is a scan line based incremental channel

router. The spacing between the prerouted wires is left as stretchable. When a new net is routed, the exact location is determined to minimize some penalties, such as distance, switching between layers, increasing channel size, and running adjacent to noisy or sensitive layers. Once the optimal path is found, routing that net pair is completed and the router proceeds to the next net pair.

The channels in SALIM [39, 40] are obtained automatically from the slicing tree description of the floorplan. Thus, global routing is a simplified form of maze routing which only tries to find the best sequence of crossed channels to minimize the length of each interconnection. Detailed routing, on the other hand, completes the routing in each channel under geometrical and electrical constraints. Among the electrical constraints are low resistance paths for power supplies, minimum number of crossings for signal paths, assigning routing priority for sensitive nets, and abiding by symmetry requirements. Most of these constraints cannot be determined by the tool and are thus provided as rules by the user. An alternative router for SALIM [41] is gridless and uses electrical constraints before the design rules. The routing strategy is again a two-step router, where symbolic routing is carried out to obtain zero-width tracks, whereby electrical parameters can be extracted and constraints can be met as much as possible. Once crosstalk, resistivity, capacitance to ground and electrical symmetry constraints are satisfied, detailed routing is carried out to fulfill design rules. In performing the placement, SALIM uses expert information to place critical blocks. This information is also utilized for ordering the connections such that wiring is done in exactly the same order as the placement. Electrical symmetry is again achieved by the use of expert knowledge. Detailed routing follows the symbolic routing to complete the routing design.

SLAM [29, 42, 43] uses a routing approach similar to MIGHTY. However, it performs a critical net analysis to determine various node types, such as noisy or sensitive nodes. A distance constraint is assigned for each circuit node with different priorities during this analysis. The circuit node with the highest-priority distance constraint should require the shortest wire connections. In addition, a prioritized spacing constraint for each pair of circuit nodes is also provided. The spacing constraint between the sensitive and the noisy nodes will get very high priority. Therefore, large spacing is reserved for those nodes to minimize the wire crossover or adjacent wire crosstalk.

The proposed global router in [44] works in a hierarchical fashion, initially creating a slicing tree deep enough such that each module is left alone in its appropriate box. The "hierarchical channel graph" thus constructed is utilized to determine the routing areas, which are essentially the spaces between the modules. An approximate rectilinear-Steiner-tree (RST) algorithm is applied iteratively up the hierarchy to obtain the routing. One should also note that the pins for combinations of modules in the hierarchy are not real pins, but pseudo pins, which should be assigned and extracted hierarchically. Nets are classified into several categories, such as sensitive or noisy nets. Special care is taken to obtain net-crossing-free routing for noisy nets. This is achieved through traffic light routing algorithm and terminal grouping. Net ordering based on the categorization of the nets is also carried out.

A plan-based layout algorithm was proposed in [45, 46]. A design plan was extracted from a circuit by using AI techniques and hierarchy information for layout generation. Initially, placement and orientation of the modules was achieved. Then, the problem of routing was tackled. The possible routes were classified into two terminal routes and multiterminal routes. These routes were ordered in a list based on their sensitivity, on their distance, on the existence of constraints coming from the knowledge base, or on whether they were straight or formed intersections with other nets. The nets were fetched from these ordered lists in sequence and candidate paths were identified. Thus, a list of option paths was produced and ordered, so that if needed the system could suggest alternative paths without having to repeat the whole process. It should be noted that the paths evaluated were treated by the system as only a set of suggestions, as the solution could not be guaranteed until all the conflicts were removed. One interesting feature of this layout generation algorithm was the layout representation used. The data were anchored to a virtual grid. The final layout was thus easily converted to a stick-diagram-like representation, which could further be converted to an ordinary layout. This strategy was suggestive of more recent template-based routing approaches.

4.5.2 Routing Based on Cost Minimization

The limitations of the digitally inspired routing methodologies, due to heuristic estimation of costs were soon obvious. The late 1980s and early 1990s witnessed the development of more "analog" routing approaches. The common themes in these routers are *area routing* and the minimization of some *cost function*. ANAGRAM [47], ANAGRAM II [21], and the area router in [30] are examples to these routers. *Area routing* is used with any class of circuits and geometric complexity as opposed to channel routing inherited from digital design automation. Since its cost function can be built as a target function for a multiple objective optimization problem, it is very flexible. The major drawback of area routing, which is its time complexity, can be drastically reduced by means of heuristic techniques. The *cost function* is typically composed of capacitance to ground for a certain node or inter-node capacitances or wire resistances. Thus, there is no explicit link between the performance constraints of the layout and the cost that the router is trying to minimize. It was assumed that individual minimization of these costs would result in a satisfactory layout solution at the end.

In ANAGRAM, the routing engine utilizes a line expansion style router, which models crosstalk directly. A uniform grid-based routing graph is defined over the entire routing space, where a vertex defines a partition on the wiring space and an edge defines a wire segment. A cost function is defined on each edge of this graph and is associated with the represented wire. The cost associated with a route can be given as:

$$\begin{aligned} Cost = {} & Cost(P) + MaterialCost(C) + ParasiticCost(C) \\ & + RoutabilityCost(C) + CostToTarget(C, T) \end{aligned} \tag{4.4}$$

Cost(P) is the cost to reach this cell from the source, *MaterialCost*(C) accounts for the incremental length added by cell C and the cost of routing on C's layer or via, *ParasiticCost*(C) accounts for the incremental parasitic to each interacting nearby net, *RoutabilityCost*(C) estimates how difficult it may be to embed this cell in this region of the layout (wire crowding), and *CostToTarget*(C, T) is a lower bound on the cost of the remainder of the path (to be estimated). The cost of a path on the graph is defined as the sum of the costs of the edges in the path. In this way, the routing problem is reduced to the search of a minimum cost path. The complexity of this search grows as a quadratic function of the circuit size. Each net is composed of a set of partial paths, which have a cost associated with them and are stored in a cost ordered structure. The next partial path to be extended (by a segment) is the one with the lowest cost. The line-expansion router thus operates by repetitively popping the most promising partial path from a heap,[5] expanding lines from the front of this partial path to the next interesting feature in the layout, and adding these new paths to the heap. A partial path here is simply a collection of connected wire segments. The *CostToTarget*(C, T) ensures that the search is biased toward the target. This routing style is especially effective when routing from distributed terminals, which occur frequently in analog layouts, such as the perimeter of a capacitor plate or the terminal of a module consisting of several transistors. Multiterminal nets can also be routed in this manner. Evolving routes are penalized according to their coupling with the previously routed wires. In addition, ANAGRAM uses this router in a rip-up-reroute phase to eliminate crosstalk violations resulting from net ordering. Another difference of this router from the previous ones is the direct inclusion of crosstalk into the routing. Wires are classified into three categories; neutral, noisy, and sensitive. Neutral wires are typically power supply and bias lines, whereas noisy wires are those exhibiting high swings, such as wires of clock or output nets. Using some simple equations to model the interactions between noisy and sensitive wires, costs can be calculated. Despite including the crosstalk into the initial routing, iterative rip-up and reroute can also improve the routing considerably. The main problem with ANAGRAM is that a cost function in terms of routing parasitics is used, but with no explicit reference to performance specifications of the circuit. With this approach, no provision is made for a constraint-driven synthesis approach. Net scheduling is not fully addressed, and the solution of congestion problems relies on an aggressive rip-up and reroute scheme.

A more advanced version of ANAGRAM; namely, ANAGRAM II was developed later [21]. Algorithmically, ANAGRAM II still employs a line-expansion strategy. However, the original ANAGRAM router used a coarse-gridded representation that limited its ability to handle over-the-device wiring and arbitrary-width wiring. Moreover, it had no support for guaranteed symmetric wiring of differential paths. ANAGRAM II was designed to address these particular limitations. In addition, ANAGRAM II can support a more interactive style of routing as desired by the

[5] A heap is a specialized tree-based data structure such that the smallest element is always in the root node [48].

user. The major difference between ANAGRAM II and the original ANAGRAM router is that the path-segments explored during line expansion in ANAGRAM II are actual rectangles of arbitrary shape rather than the line segments on an abstract grid. In ANAGRAM II, the *RoutabilityCost*(C) term measures the cost associated with ripping up nets in the neighborhood necessary to advance a path; ANAGRAM II thus makes an explicit tradeoff between detouring around an obstacle, and simply removing it (if its cost was sufficiently low) to reschedule it for later rerouting.

In [30], a modified version of Lee's algorithm is proposed for routing. A cost function is formed for each net based on the weighted distance between nodes, a layer resistivity parameter, a layer-to-bulk capacity parameter, proximity parameter dependent on the distance of the node from the already existing wires, and a congestion parameter, based on the real wire crowding of the surrounding area and on an estimate of its final crowding based on a fast first-attempt path-search for each of the remaining wires. The weights corresponding to each component of the cost function is set by the user on a net basis. Backtracing on the wires is also performed to clean up extra corners. The approach also accommodates preconnected pins, such as those in a stacked transistor structure. Symmetry is considered as an issue in [30] as well. The routing system also proposes a scheduler, which determines the routing order of the nets. The scheduler utilizes symmetry information, user information, and congestion information to arrive at a priority list. Obviously, the scheduler does not solve the ordering problem completely and provisions for rip-up and reroute are also present.

4.5.3 Routing Based on Parasitic Bounds

The routers above including ANAGRAM II minimize crosstalk, but without any specific, quantitative performance targets. On the other hand, the routers ROAD [19] and ANAGRAM III [49] use improved cost-based schemes that route instead to minimize the deviation from acceptable parasitic supplied by the designers or derived from sensitivity analysis. In Fig. 4.18, routing flow for these routers is depicted.

4.5.3.1 Constraint Generation and Sensitivity Calculation

As discussed above, the link between the performance of a circuit and the parasitic bounds on a node can be determined either by the designer or by the sensitivity analysis, which acts as a constraint generator. The process of constraint generation is not trivial and is discussed in detail in [50, 51], and [52]. For meeting the performance specifications of a circuit, finite degradation can always be allowed in the performance functions during routing, as long as they are below certain thresholds. The constraint generator is used to map a set of high-level performance specifications onto a set of bounds, which are then used during the layout synthesis to control parasitics. The performance constraints during routing are the maximum

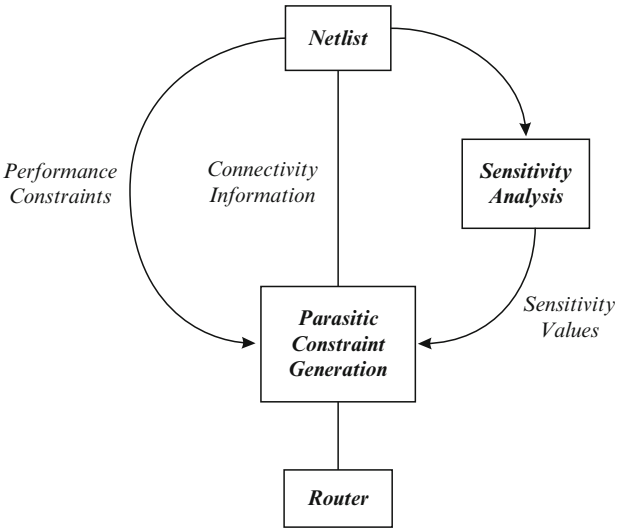


Fig. 4.18 Sensitivity based Routing

changes allowed in performance functions because of routing parasitics. All possible combinations of interconnect parasitics, which meet performance constraints define a feasible region in the space described by the parasitics, which in fact can be treated as the design parameters for routing. Hence, analog routing can be treated as a constraint-driven optimization problem. The objective function is the chip area, and the constraints are the performance constraints of the circuit. The parasitics considered can be line resistances, capacitances, inductances or line-to-line capacitances and inductances. The parasitic constraints imposed on the router are of two types: bounding constraints and matching constraints. Based on the performance sensitivities, performance constraints, and a-priori estimates of maximum values of parasitics, the set of parasitics, which can collectively cause negligible performance degradation, are ignored for generation of bounding constraints. The rest of the parasitics on which bounding constraints are imposed are called the critical parasitics. Please note that in general there will be many possible combinations of bounding constraints, which meet performance constraints. These constraints can not only be utilized for area routing, but also in modifying vertical constraint graphs in channel routing for mixed signal circuits [53, 54]. The sensitivity S_{ij} of a performance function W_i with respect to a parasitic p_j at the nominal value of W_i is defined as;

$$S_{ij} = \left[\frac{\partial W_i}{\partial p_j}\right]_{p_j=0} \tag{4.5}$$

One way to compute sensitivity is by using the perturbation method. In this method, a parameter is perturbed and the performance is reevaluated using circuit simulation. However, this method is very time consuming since a re-simulation of

the circuit is required for each parameter of interest. This method can also be very error-prone, particularly for transient simulations, since the circuit simulator output has some inherent error, and while taking difference between two close numbers, the percentage error gets larger. Sensitivities can be computed much more efficiently and accurately, compared to the perturbation method, by using direct or adjoint techniques of sensitivity computation. A similar approach can be used for matching constraints as well. Once all the sensitivities are calculated, approximations to performance constraints can be modeled by the following inequalities:

$$\sum_{j=1}^{N_p} S_{ij}^{+} p_j \leq \Delta W_{i_{\max}}^{+}, \forall W_i \in W^{+} \tag{4.6}$$

$$\sum_{j=1}^{N_p} S_{ij}^{-} p_j \leq \Delta W_{i_{\max}}^{-}, \forall W_i \in W^{-} \tag{4.7}$$

where N_p is the number of parasitics and

$$S_{ij}^{+} = \begin{cases} S_{ij}, & \text{if } S_{ij} \geq 0 \\ 0, & \text{if } S_{ij} < 0 \end{cases}, S_{ij}^{-} = \begin{cases} -S_{ij}, & \text{if } S_{ij} \leq 0 \\ 0, & \text{if } S_{ij} > 0 \end{cases}$$

Terms $\Delta W_{i_{\max}}^{+}$ and $\Delta W_{i_{\max}}^{-}$ are the maximum allowed change in W_i in the positive and negative direction due to parasitics, respectively. Similarly, terms W^{+} and W^{-} are the set of performance functions having constraint in the positive and negative direction from nominal, respectively.

Please note that the above inequalities have infinitely many solutions and it is quite difficult to select the "best" solution among these. That is, one can obtain many different sets of bounds on parasitics p_j starting from the bounds on W_i. Furthermore, for any practical circuit, the number of parasitics and thus sensitivities are quite large. A simple thresholding technique or Independent Component Analysis can be applied to the sensitivities to simplify the problem since many parasitics have very little effect or no effect at all on some of the performance metrics. Thus, the dimensionality of the problem gets much smaller. Finding the actual solution is still quite difficult thus requiring heuristics such as sensitivity graphs [55], which can be simplified and later used by the router or flexibility values for parasitics [31].

The parasitic generator PARCAR defines a flexibility value for each parasitic in addition to the above. This value is calculated from the minimum and maximum values for the parasitics [31]. The maxima and minima for each parasitic are generally not known before the layout is drawn, but can be initially estimated. As the layout evolves, the flexibility values get more accurate. The flexibilities can be used inside a quadratic programming package. To ease the solution, the parasitics that are not very effective are ignored. As a result, less flexible parasitics are tried to be satisfied with more effort, whereas the layouts for more flexible ones can be more easily drawn.

4.5.3.2 Routers Based on Parasitic Bounding

The router in [56] runs in conjunction with a constraint generator, which computes the sensitivities of performance functions to all possible parasitics in the circuit and detects the critical parasitics. It also generates a set of bounds on critical parasitics to satisfy performance constraints. In differential circuits, a set of matching constraints on parasitics based on matched-node-pair information is evaluated, and worst-case sensitivities are computed taking mismatch into account. Sensitivities are used to generate the weights for the cost function driving the router. The router itself is a maze router using the A* algorithm (A* algorithm is discussed at the end of this section). The cost of a path is calculated not as a direct parasitic cost, such as the total capacitance of the path, but as the cost it has on the performance of the circuit by using the weights obtained from the sensitivity analyzer. Thus, the quality of the routing is dictated by the accuracy of the weights. Linear approximations for sensitivities are acceptable, because the goal is to keep each performance within a small tolerance specified by the user. The way parasitic bounding constraints are generated is such that if the value of each critical parasitic remains within its bound, then all the performance variations remain within their respective bounds too. Therefore, if a performance constraint is violated, at least one parasitic must have exceeded its bound as well. Hence, one possible approach to modify the weights automatically is to increase the ones associated with the parasitics whose values resulted larger than the bounding constraints and to decrease the ones associated with the parasitics whose values were smaller. But with such an approach weights could oscillate indefinitely through iterations. The authors also present a heuristic method to adjust the weights as well during the routing.

A router for analog design (ROAD) is a maze router based on the A* algorithm, using a nonuniform grid with dynamic allocation. It allows over-the-device routing, although routing over sensitive modules can be prevented. Two operating modes are available. In interactive mode, the user can either accept the routing order suggested by a scheduler, or define a different order and modify the circuit configuration with rip-up and reroute operations. In batch mode, a routing session can be programmed for execution without requiring the user's attention. Automatic routing scheduling or a predefined or partially defined order can be used. At the end, a comparison between parasitic values and upper-bound specifications is performed and decoupling shields are built where necessary. At the same time, interactive mode provides expert analog designers with high flexibility not contrasting with full automatic features. The routing graph used by ROAD is a three-dimensional nonuniform grid; the grid is further refined to reduce the maximum size and aspect ratio of rectangular area portions. Every new wire determines a local grid refinement and the dynamic allocation of new nodes. On the grid edges, wire segments are generated and the cost function is computed. However, the grid does not constrain the wire size, pitch, or position as fixed, or virtual grids do. In fact, local congestion and in general all the parameters of the cost function are computed with respect to the whole space locally available. As a result, the router achieves a complete control over all the

details of routing geometry. Terminals and blockages can be arbitrary collections of geometries. Over-the-device routing and crosstalk sensitivity analysis to pieces of placed devices are possible without additional overhead, as required by sophisticated data structures. The router ROAD can be used in conjunction with PARCAR to complete the routing. In ROAD, the cost function is a weighted sum of several nonhomogeneous items. These are local area crowding, resistance, capacitance to bulk, and cross capacitance. Performance sensitivities to parasitics are used to generate the weights for the cost function driving the area router. The contribution of a parasitic to performance degradation is proportional to the sensitivity and inversely proportional to the maximum variation range allowed for that performance. The routing schedule is determined with a set of heuristic rules set up and tuned with experimental tests. The higher the number of constraints on a net, the higher is its priority. The number of properties that a net can have, for instance symmetry, membership to supply or clock nets, etc. has already been defined. After performing the weight-driven routing, parasitics are extracted and performance degradation is estimated and compared with its specifications. If constraints are not met, the weights of the most sensitive parasitics are raised and routing is repeated. When the weights of all sensitive parasitics hit their maximum value, iterations stop. This means that even considering maximum criticality for the sensitive parasitics, routing is not possible on the given placement, without constraint violations. In this case, the circuit placement needs to be generated again, using a wider range of variation for the detected sensitive parasitics. In ROAD, nets are split according to a heuristic that estimates the parts of a net carrying low current. The routing schedule is determined on the ground of the priority assigned to each net according to its presumable difficulty due to electrical and architectural requirements. Here, symmetric nets are given a very high priority so that they can be routed first. ROAD provides provisions for including shields. Shields are implemented into the layout as a "last resort" if no other routing solution can be found to keep away sensitive nets from each other, and are built after all the wires have been routed. Many parameters can be used to modify the behavior of ROAD. Hence, it is important to provide the designer with a user interface that allows full exploitation of its flexibility. A high-level command language, called net descriptor language (NDL), provides a user interface to ROAD. The purposes of NDL commands are to assign weights for cost function and scheduler (weight values can be directly specified by the user or automatically computed), specify symmetry requirements, or declare the nets to which the net-splitter is applicable.

As discussed above, Crosstalk-sensitive analog routers (ANAGRAM II, ROAD) must rely on some variant of maze-routing with a cost-function comprised of four terms: a material term (to minimize length, vias, bends), a crosstalk penalty (to minimize proximity to deleterious signals), a routability term to estimate how easily this net fits into the layout, and a cost-to-target predictor (to accelerate search in algorithms like A* routing). Crosstalk optimization substantially degrades the efficiency of any area router because of the overhead of checking proximity effects at the head of each evolving partial path (which is unavoidable), and because crosstalk obstacles

are very hard to predict and frequently require deep path search to avoid. ANAGRAM III alleviates this problem by pruning parasitically nonviable paths as the search progresses. The virtue of this scheme is that it does not make artificial tradeoffs between wirelength and crosstalk; instead, it can efficiently find the cheapest path that does not violate hard parasitic bounds supplied by the user.

A* Algorithm

A* algorithm [57] is a general methodology for the shortest path calculations in graph theory. In [58, 59], it is adapted to the area routing problem and used to improve the average run time of the Lee—Moore algorithm. The upper-bound for its run-time is $O(n^2)$, as for the Lee—Moore algorithm, but its average performance may be much better. This algorithm operates by making estimates for the cost of connections before committing them and runs on a routing graph $G(V, E)$, extracted from the layout. In A* algorithm, path searching continues till a path from source to target is found or all vertices in V are visited. During this search, costs are assigned to vertices and these costs and the corresponding vertices are stored in an ordered list. This path search is performed in a loop and the vertex having the minimum cost is chosen to continue with. The cost of vertex x is $f(x)$ and it is formulated as follows:

$$f(x) = g(x) + h(x) \tag{4.8}$$

where $g(x)$ is the cost calculated from s – source wire – to x – an intermediate wire – and $h(x)$ is the estimate of the minimum cost from x to t – target wire –. Note that $h(x)$ in $f(x)$ must not overestimate the distance to the goal. If only the wire lengths are considered, a preferable heuristic for $h(x)$ is the Manhattan distance from x to t. In Fig. 4.19, these costs are simply depicted, where the total length of the solid lines represents $g(x)$ and the length of the dashed line represents $h(x)$. In analog routing, $h(x)$ may be defined as the resistive cost of the Manhattan path from x to t and $g(x)$ as the weighted sum of the additional parasitics cost due to the new path from

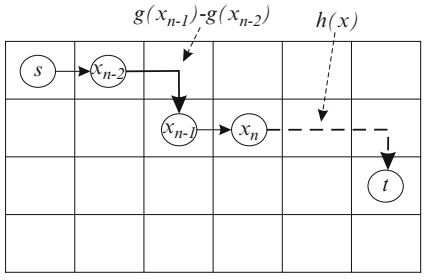


Fig. 4.19 A* algorithm uses estimates from candidate nodes x_n's to target node

s to x. If only resistive and capacitive parasitic cost are considered, $g(x)$ may be calculated as:

$$g(x) = A \cdot (G_{x-1} + \Delta G_x) \tag{4.9}$$

$$g(x) = \begin{bmatrix} a_1 \cdots a_n \end{bmatrix} \left(\begin{bmatrix} g_1 \\ \vdots \\ g_n \end{bmatrix}_{x-1} + \begin{bmatrix} \left(\sum_{\forall j} c_{ij1} \Delta C_{ij} \right) + (r_1 \Delta R) \\ \vdots \\ \left(\sum_{\forall j} c_{ijn} \Delta C_{ij} \right) + (r_n \Delta R) \end{bmatrix} \right)$$

where A is a vector including the weights for different performance metrics, G_{x-1} is the sum of the performance metrics from s upto x and ΔG_x is the change in the performance metrics due to adding x. ΔC_{ij} is the additional capacitance value between net i – net of x – and net j – net of a neighboring wire or bulk – and ΔR is the additional resistance value. Changing the coefficients c_{ijn} and r_n, it is possible to define different metrics, such that if $r_n = 0$, only capacitive parasitics are considered and if $c_{ijn} = 0$ only the resistive parasitics are considered.

As previously mentioned, the $f(x)$ costs are needed to be stored in a sorted manner, such that the algorithm chooses the wire with minimum $f(x)$ cost and continues path search. Thus, a sorted list of $\{f(x), x\}$ pairs is needed, where the sorting must be according to the f(x) values. A red–black tree[6] implementation may be used to keep the $\{f(x), x\}$ pairs ordered. However, different vertices x_i and x_j may exist such that $f(x_i) = f(x_j)$; in such a case, the tree representation will not be capable of storing $\{f(x_i), x_i\}$ and $\{f(x_j), x_j\}$ pairs. Storing $\{f(x), L_{f(x)}\}$ pairs instead of $\{f(x), x\}$ pairs solves the problem of duplicated $f(x)$ costs, where $L_{f(x)}$ is a linked list of the nodes having the same $f(x)$ value. This data structure is depicted in Fig. 4.20 where the nodes of the red–black tree point to the linked lists, associated with them.

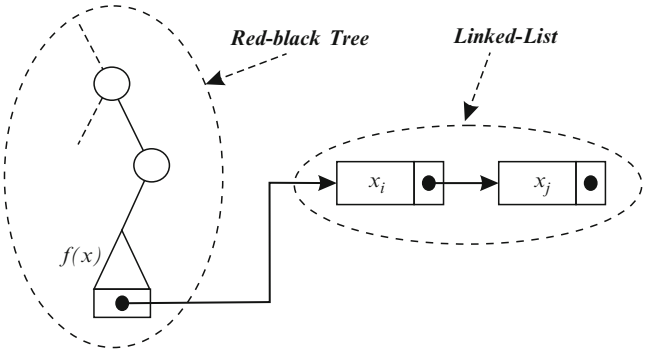


Fig. 4.20 Data structure to store $\{f(x), x\}$ pairs

[6] A red–black tree [60] is a binary search tree, it inserts and removes nodes intelligently to ensure that it is balanced.

Fig. 4.21 Pseudo Code of A* Algorithm

```
function findPath(sources, targets)
  // initialize sources
  For every source in sources
    source.G = zeros(n) // a vector of n zeros
    source.previous = null
    h_source = calculate hCost(source)
    // calculate A dot G
    g_source = calculate gCost(source.G)
    f_source = h_source + g_source
    add {f_source, source} into openSet
  // search for a path
  while(openSet not empty)
    x = the first node in openSet
    if targets contains x
      return construct_path(x)
    // get neighboring space(empty cells)
    for every y in getNeighbors(x)
      if closedSet contains y
        continue
      add y into closedSet // visit ones
      vectorG = calculate_deltaG(x, y) + x.G
      if openSet not contains y
        y.G = vectorG
        y.previous = x
        h_y = calculate_hCost(y)
        g_y = calculate_gCost(y.G)
        f_y = h_y + g_y
        add {f_y, y} into openSet
      else if calculate_gCost(vectorG)
              < calculate_gCost(y.G)
        y.G = vectorG
        y.previous = x
        h_y = calculate_hCost(y)
        g_y = calculate_gCost(y.G)
        f_y = h_source + g_source
        update f_y of y in openSet

function construct_path(x)
  if x.previous == null
    return path
  else
    add new wire from x to x.previous into path
    construct_path(x.previous)
```

The pseudo code for the A* algorithm is given in Fig. 4.21. In the code, two sets are used, namely, the closed-set and the open-set. The closed-set, a hash set, contains the visited vertices and the open-set, an ordered set with the structure in Fig. 4.20, contains the $\{f(x), x\}$ pairs. In the pseudo code, `calculate_hCost(y)` estimates the resistive cost from y to target and `calculate_gCost(y.G)` takes the product of the input vector with the weighting vector A.

4.5.4 Integrated Placement and Routing

Conventionally, the execution of placement and routing has been sequential. If the routing is a two-step procedure, the execution of the global routing and detailed routing is also sequential. Thus, the global routing can be beneficial only for the

detailed routing and it is of little use to the placement procedure. Normally, only a rough wire-length estimator is employed during the placement. So the output placement solution is likely not an optimum as viewed by the global routing. Although the simultaneous performance optimization of the placement and the global routing can lead to a more accurate search, the computation time is impractically high when applied to large digital circuits due to huge number of nets to be considered. However, compared to digital circuits, the number of nets in analog integrated circuits is relatively small. This allows the global routing to be considered along with the placement procedure. On the other hand, the performance of the analog layout is very sensitive to the actual wire paths. Bringing the global routing into the placement will ease the implementation of those parasitic constraints. The integration of placement and routing can be done at several levels of granularity. On the one extreme, routing can be implemented between the iterations of placement. This approach does not require much change in the existing placement and routing algorithms. On the other extreme, routing can be viewed as a placement of the connecting wires. Thus, placement and routing are merged into a single enhanced placement step.

A novel idea incorporated into the KOAN/ANAGRAM II system is to extend the annealing-based KOAN placer so that it can manipulate both devices and wires [61]. This is a very early example of simultaneous placement and routing and can alleviate the need for separate complex routing algorithms. A simple routing solution can be improved over time together with the placement. The central problem is how to represent fully detailed wire geometry in a manner that allows the same freedom of incremental movement as the devices themselves. The simulated annealing algorithm of KOAN can place wires just like placing modules except for three properties; the representation, the moves, and the cost function. The nets should be represented such that they are always electrically connected correctly to their corresponding devices. The feasibility or quality of connections is not important initially as they will improve over time. It is enough for the router to find just a connection between the devices. Also note that as the devices are moving, the wires are being constantly reshaped. Not much thought is given into the allowable moves, but the effort has been transferred into careful design of the cost function that coerces nets to evolve into high-quality physical routes. The cost function includes design rule violations, crosstalk, and total net area for one connection.

RACHANA [62] also describes an integrated placement and routing approach. Initially, a graph of the circuit is created based on the schematic. This relies on the widespread assumption that the best distribution of the elements is already present in the schematic if the schematic is drawn properly. Then, constraint-driven module generation is carried out. Some modules have been coded into RACHANA such that it can recognize these simple subcircuits and create many layout variants for them. In the third phase, which is the floorplanning phase, the best configuration among the variants is selected. In the unified placement and routing algorithm, the placement and routing steps are simply intertwined. In other words, a module is placed, then the connections to that module are completed before the next one is placed. The router itself is gridless and multilayer.

The developers of GELSA [63, 64] claim that the integration of placement and routing can be achieved by doing routing at every step of the placement process. This in turn results in the solution of the routing problem, which is NP complete, at every iteration of placement, thus resulting in extremely complex layout generation. To avoid excessively long run times, the routing is not completed for every solution at every step, but approximate routing is carried out. Slicing structures are used for problem representation, whereas the optimization algorithm is simulated annealing. The slicing structures are simply shown by a tree or a string. The optimizer also takes symmetry into account. The approach for symmetry is quite interesting in that two levels of symmetry are simultaneously considered in the cost function: global symmetry with respect to virtual axes, and local symmetry affecting groups of cells.

In [65], the possibility of simultaneous placement and routing is also explored. The sequence pair algorithm is utilized for the placement of blocks as well as wires. Each wire is divided into a set of rectangles, and the following two extensions are introduced to maintain the connection: one is to impose a condition of orders of rectangles on a sequence-pair called wire-connectivity, and the other is to generate horizontal and vertical constraint graphs for compaction.

The potential problems of the conventionally separate placement and global routing in analog integrated circuits, which often involve complex constraints, were also addressed in [66]. This work presents a two-stage placement technique to solve the analog macro-cell placement problem. The entire placement procedure is divided into global placement and detailed placement stages. During the global placement, a hybrid genetic placement approach using a half-perimeter wire-length estimator is employed. It performs a rough but quick search to locate the region of the optimum. In the detailed placement, a very fast simulated reannealing placement approach and a minimum-Steiner-tree-based global routing are executed simultaneously. In this manner, the optimum can be found by searching a relatively small region. For each intermediate placement solution, the global routing elaborates the routing plan, taking into account the net sensitivity and channel congestion. Moreover, the cost obtained by the global routing is used to evaluate the quality of a placement solution. Thus, the placement solution with the lowest cost (i.e., the optimum in terms of the global routing) will be sought when the optimization process progresses.

Simultaneous placement and routing was extended even further to integrate constraint transformation into the integrated place and route as well [67]. The circuit was represented geometrically as tiles. The tiles could be moved, swapped, routed, and resized. A tabu-search optimization algorithm was utilized with these available steps. The end results were shown to be superior to conventional sequential constraint transformation-place-route methodology.

4.5.5 Global/Detailed Routing

In ALADIN [68, 69] two routing phases, namely, global routing and detailed routing are employed. The global routing is integrated into the placement procedure

to improve the accuracy of routing estimation. The compaction-based constructive detailed routing generates the final layout based on the output of the placement procedure. Because nets play a critical role in analog circuits due to parasitic effects, crosstalk, etc., minimum-Steiner-tree-based global routing is developed. The estimation of net-length is critical for the placement and the global routing. The choice of a suitable net-length estimator is actually a tradeoff between the accuracy and the computation efficiency. In ALADIN, several typical net-length estimators, including the half-perimeter, the center-of-mass, the complete graph, and the minimum spanning tree have been developed. All these methods are based on Manhattan distance, which inevitably degrades the reliability of the estimation. Since nets play a critical role in analog circuits due to parasitic effects, crosstalk, etc., this problem is addressed by developing a method based on the minimum-Steiner-tree. Not only is the minimum-Steiner-tree method used for net-length estimation, but it also elaborates the routing plan, taking net sensitivity and channel congestion into account. A weighted graph is used to model the routing regions. A rectilinear channel graph is formed by passing channels (or edges) through critical regions and forming vertices at their intersections (Fig. 4.8). Finding a global route becomes equivalent to finding an optimal subtree (the minimum-Steiner-tree) in the routing pin graph that spans the terminal vertices. A Dijksta shortest path algorithm [70] is applied to solve this minimum-Steiner- tree problem.

A technique of simultaneous execution between the placement and global routing has been developed in ALADIN, where the global routing is executed for each intermediate placement solutions. It makes better search results without losing the solvability of the problem. This global-routing-driven placement strategy is especially effective for the analog layout designs, where the number of nets is relatively small but with complex constraints. A potential problem in the traditional placement and global routing procedures is whether the estimated channel width is accurate. So a postprocessing procedure, such as compaction, is required. However, a different strategy is used in ALADIN, where the placement phase is followed by a compaction-based constructive detailed routing phase that automatically minimizes the channel space. The width of channels need not be considered during the placement except for the channel congestion to avoid overburdened channels. The congestion degree of a channel is represented by the number of the passed nets in this channel. It is taken as a weight in the weighted graph of the global routing model, apart from the channel length. The Dijksta shortest path algorithm optimizes the routing paths and finds the one that is balanced and the shortest. In this way, the conventional problems of routability and postprocessing are avoided. The global routing is integrated into the placement procedure to improve the accuracy of the routing path estimation . The compaction-based constructive detailed routing completes the final layout based on the output from the placement procedure. In the detailed routing phase, an initial preprocessing step is applied to determine the order of the wiring. This preprocessing is heuristic and starts from symmetric modules and more central modules. After ordering, for each module, the interconnections within the module are first wired densely around the module boundary using a ring router [71]. Then, this dense module is compacted toward others according to the

position relationship extracted from the placement solution. The interconnections between the compacting module and the reference module are routed within a relatively small scope using a modified maze router [71]. In this way, the whole layout area remains as small as possible because the compaction can assure high density. Moreover, no estimation of routing area is needed and the problem of routability can be solved during each compaction step.

The concept of symmetric routing was applied to net bundles, which may occur both in digital and in analog designs in [72]. Previous discussions on differential capacitance in net bundles was limited to digital design and consisted of simple Miller effect-based approximations. In [72], the net length difference due to corners in the routing of bundles is discussed. Furthermore, the slightly different effect of the aggressor on each individual net in the bundle is investigated. The methodology presented tries to minimize these effects by proper choice of the routing area. If this is still not enough to balance the nets, extra routing paths and adjustment sections are added.

In [73] and [74], a two-step routing strategy is suggested. A channel intersection graph model is employed to represent the global routing structure such as in Fig. 4.8; thus, a routable layout region is confined to intersection regions of the modules. The routing model employed is also coherent with the slicing floorplanning model used for placement.

Routing connections of the modules are projected on the enclosing routing channel as the first step of routing. The problem of global routing then reduces to connecting the points on the intersection grid of the same node. An iterative maze routing algorithm is employed for global routing. The implemented algorithm is sequential; one connection is routed in a single iteration. Exploration of the maze router is based on backtracking algorithm, so that all possible routing solutions are explored. Although backtracking is very time-consuming, most of the branches that exceed performance bounds are eliminated during routing. Physical routing is not finished at this point but, it is possible to predict node-to-node and node-to-ground capacitances using global routing information. Thus, it is possible to calculate the effect of each routing option on the performance using predicted capacitances and sensitivity information. Minimum performance degradation is allowed for all iterations of the maze router.

The next step of routing is local routing, where the global routing information is used to generate the actual physical routing. Routing information registered on the intersection graph is mapped onto channels that are located between or beside the modules. The results of mapping the intersection graph to the channel indicate directions in which routing should be performed. The channel devoted for routing is generated through four basic operations. These operations produce a channel route plan clearly identifying the structure of detailed routing. A connection is inserted into the channel via a "push" operation, while it is propagated to the next connection point by "propagate," redirected by "peak," and directed out permanently by "pop." Routing ends when all of the connections of the same label are connected. A refinement step is run over the resulting plan to reduce transitions between routing levels, producing a smoother routing.

One more step is necessary to produce final routing. The final routing step used to complete routing employs switchbox routing scheme to physically produce routing. Switchboxes are inserted at each position of the channel where a level transition occurs. Given the node route depths and route directions, switchboxes generate the detailed routing of the channel. Space between the switchboxes is simply filled with connecting wires. Combining all switchboxes and interconnections between the switchboxes, results in complete layout of the channel. Similar to the optimization method utilized in global routing, local routing also uses a backtracking-based exhaustive algorithm. Possible switchbox configurations are enumerated to achieve coupling information. Note that a typical switchbox does not exceed a size of 3-by-3; thus, computation time is not a critical criterion for this step.

4.5.6 Template Based Approaches

Many layout tools do not generate the layout by themselves, but they use the information provided by the user. This information may be in terms of a sample layout or may be coded in a specific layout language. Although this approach may seem to be much easier and much more primitive compared to the previously discussed routers, it has many applications. The general reluctance of analog designers toward using automatic layout generation tools is because they are skeptical about their success/reliability and they would like to have full control over the layout generation process. They would prefer the layout tool to be an assistant to them rather than producing the layout itself. Thus, generating a template for the layout is quite a good solution for them. Furthermore, in many cases, a designer may desire to port a layout created in one technology to another similar technology. The designer may also want to make an incremental change in the design after the layout, while remaining in the same technology. Automatically generating the layout from scratch for these problems is not only costly, but may also create inferior results. Since most layout generators contain probabilistic components, the generated layouts from the same circuit will not be the same for every run of the generator. Thus, when the designer has a successful layout with all the parasitics estimated, he/she will not want the tool to create a new layout from scratch, but will want an update of an existing design. In these cases, generation of a layout from a template with preextracted parasitics will be much more desirable.

One of the earliest proposals for such a layout generation approach was in [75]. This approach uses a sample layout, the template, to graphically capture an expert's knowledge of analog device placement and routing for a given module type. To generate a module, one supplies the required electrical parameters for each device and a geometrical constraint on the module's shape e.g., a desired aspect ratio. Using exhaustive floorplan area optimization techniques, the tool then determines the optimum shape of each device to satisfy the user's geometrical constraint. Subsequently, the layout is generated by transforming (via compaction) the template into a module, substituting the devices in the template by newly generated devices with the

user-supplied electrical parameters and the determined geometrical shapes. This technique produces good quality layout in a reasonable amount of time, by utilizing the expert designer knowledge embedded in the template and by taking analog specific features such as device matching and merging into account during the layout transformation phase. The routing in the template remains intact while the modules are interchanged with the desired ones. The only job of the router, if one may call it so, is to stretch or compact the interconnections according to the new module sizes.

The template may also be described with a layout language, such as BALLISTIC [76]. Layouts represented with this language can be transported across technologies easily as well as fitting into various aspect ratios. Interconnections are also represented by a wiring command, which is relative just like the placement commands. The interconnection is described via the layers utilized and the breakpoints in the wire. A few hundred lines of code are enough for a medium complexity analog cell. This code is then translated into the native layout code of a commercial design tool.

Another language proposed in this manner (CAIRO) is composed of a documented superset of C functions [77, 78] rather than a specific language. Then, this code can be compiled to generate the layout. The routing functions allow relative routing description using predefined reference points. This results in a shape-independent description of the routing.

The authors of [79, 80] propose a template-based approach for retargeting. The proposed retargeting methodology relies on the previous existence of a block netlist, layout templates for the block at hand, and, optionally, some tuning strategies in the form of design constraints for such blocks. When parameterizing complex layout cells, factors such as regularity, density, and symmetries are kept during the retargeting process. This is achieved by relying on a deep hierarchical decomposition and a careful cell planning. Parameterized layout templates are first built for single devices and small numbers of them (i.e., a set of matching transistors). These basic structures are used to build more complex parameterized subcells, proceeding up the hierarchy until the layout template for the objective block is obtained. During this constructive process, much attention is paid to the complete parameterization of cells, relative positions and interconnections, so that, big changes in device sizes can easily be accommodated. Parameterization of the interconnections does not only consider the design rules but also the current densities that must be carried. The parameterized layout templates have been built using a commercial tool for easier acceptance by analog designers.

In LAYGEN [81, 82], the router uses the placement solution and the template's nets to produce the desired routing. The algorithm uses a two-stage generation process; first, it adjusts the template routing to the newly created placement, then the optimizer attempts to adapt the routing to the particular layout representation. The new placement yields new pin positions so the template routing paths must be adjusted to the new pin locations. Each net is divided in a set of wires, each one connecting two and only two pins. The adjustment procedure consists of the following: First, the template paths are scaled, then moved to set the wire start point

on the new start-pin position and, finally the wire stop position is set to the new stop-pin position to ensure connectivity. The adapted routing is then used as a start point for the evolutionary optimizer. The optimizer uses the information in the new placement and the adapted nets to minimize the cost function that incorporates design rule violations, connectivity requirements, wire length to increase area usage and to decrease parasitic capacitances, and minimum distance between nets to separate nets as far as possible reducing the crosstalk. The genetic optimizer encodes the routing information by assigning one gene to each adapted net. In this way, crossover generates children that present a combination of their parents' nets, and mutation is performed in each net. The advantage of such a complex genetic encoding is that the mutation operators can be designed to be more "intelligent" as they use more information. It is also interesting to note that the validations required in routing makes it more complex and computationally more expensive than placement for LAYGEN.

4.5.6.1 A Simple Template Script

Simplicity and reusability are the main aims of a template. A simple template script, called layout description script (LDS), will be described in this subsection. The simplicity of this script comes from its representation, where absolute or relative positions of layout elements are defined as simple equations. These layout elements may be modules (transistors, capacitors, etc.), wires, or even wells. Top, bottom, right, and left boundaries of these elements are used in the description. For instance, the LDS code for symbolic layout in Figs. 4.22a and 4.22c are defined in Figs. 4.22b and 4.22d, respectively. Note that the description does not include any redundant information and a code line includes only horizontal or vertical information and not both.

LDS is needed to be extracted from a placement before the script and the dimensions of the layout elements are used to synthesize new layout instances. Using vertical and horizontal constraint graphs, LDS code may be easily extracted. For the horizontal placement in Fig. 4.23a, the horizontal constraint graph in Fig. 4.23d may be used to automatically extract the code. This graph has an edge if the right boundary of an element sees the left boundary of another element. Similarly, edges of the vertical constraint graph are between the top and the bottom boundaries of the layout elements. The LDS code corresponding to Fig. 4.23d is given in Fig. 4.23f. However, adding this code into the template will not suffice to synthesize overlap free layouts, due to the fact that the horizontal and the vertical information are coded separately. In Fig. 4.23b, a layout synthesized from the code of Fig. 4.23f is shown. Here, the width of b and the height of e are enlarged and the resulting layout has overlapping elements. A way to prevent this overlap is to combine some vertical information with the horizontal information. The constraint graph in Fig. 4.23e guarantees nonoverlapping layouts for any size of elements. Such a graph is constructed by adding edges from the right boundary of an element to the left boundary of the elements that have higher x positions and have no other elements in between.

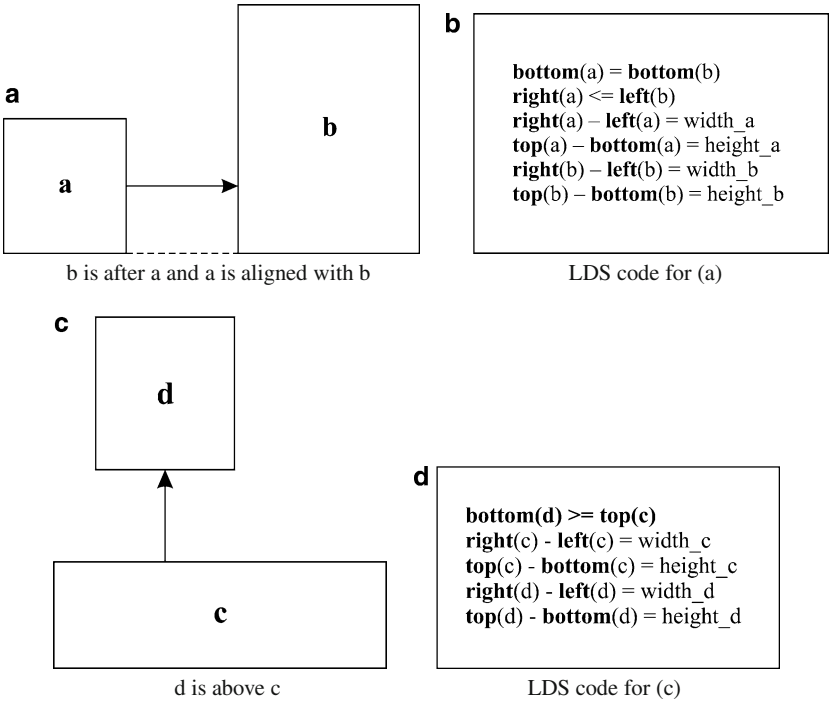


Fig. 4.22 LDS descriptions for the layouts in (**a**) and (**c**) are listed in (**b**) and (**d**), respectively

Corresponding LDS code and placement are given in Figs. 4.23g and 4.23c, respectively; note that the overhead due to the extra edges is not much because of the fact that edges are only added between the elements that may overlap. The extra space after the sizing is removed by applying a compaction. This way, the template does not need to contain any absolute position and the sizes of the modules may be freely updated. Although such a template handles overlaps, it does not handle design rules. For instance, there must be a spacing between wires of different nets if they are in the same layer (constraints for each layer are extracted separately); this space depends on the manufacturing technology. However, it should not be added between the wires in the same net. Thus, during the extraction of constraints only some of the elements are considered, such that these elements are in the same layer but they belong to different nets.

In Fig. 4.24a, three wires are shown in the first metal layer and they are connected. Extracted LDS code for the W2 is in Fig. 4.24e. This code takes care of changes on the width of W2, and the wire W3 must be after wire W1. Due to the fact that W3 must be after W1, the flexibility of the template is limited, where as the structure in Fig. 4.24b is dynamic and does not restrict the wires horizontally as shown in Figs. 4.24c, d. The structure in Fig. 4.24b uses four wires. LDS codes for W2 and W4 are in Fig. 4.24f. The constraints in these LDS codes may be solved with an LP solver.

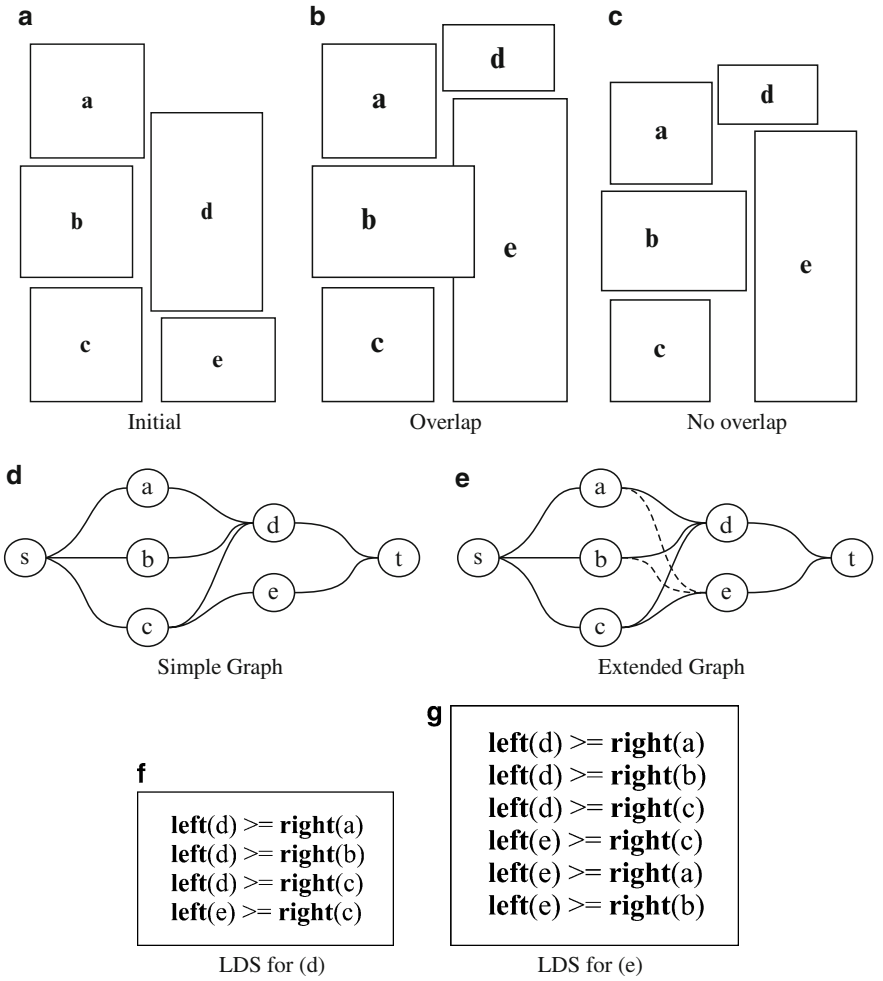
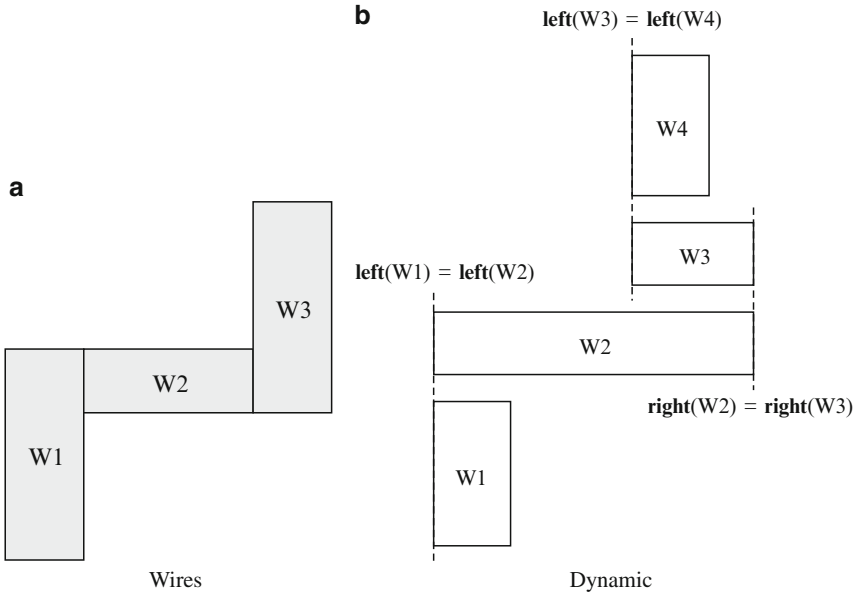


Fig. 4.23 Resizing affects horizontal and vertical constraints: After resizing, placement in (**a**) may result in an overlapping layout as in (**b**) if the horizontal graph in (**d**) and the corresponding LDS in (**f**) are used. Overlap in the layout is prevented in (**c**) when the horizontal graph in (**e**) and the corresponding LDS in (**g**) are used

Through LDS, a template is coded for the OPAMP in Fig. 4.25a. Using this template, the sample layout in Fig. 4.25b is synthesized. The same template is also used to synthesize the layout in Fig. 4.25c, however, the dimension of the M3–M4 transistor pair is enlarged. Note that the path between the M3–M4 pair and the M1–M2 pair is coded as the path in Fig. 4.24b and it is dynamic. Similarly, the same template is used to synthesize the layout in Fig. 4.25d. In this sample, the dimensions of the transistors, M6 and M7, are narrowed. For all these cases, the resulting layouts are free of overlaps and they do not violate the specified design rules such as minimum width, spacing, etc.



e

right(W2) = **left**(W3)
top(W2) = **top**(W1)
bottom(W2) = **bottom**(W3)
top(W2) - **bottom**(W2) = **minWidth**(MET1)
right(W2) - **left**(W2) >= **minWidth**(MET1)

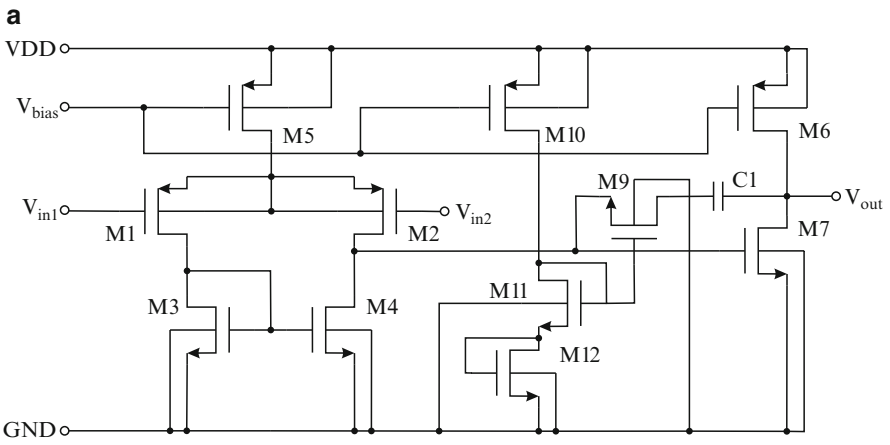
LDS of (a)

f

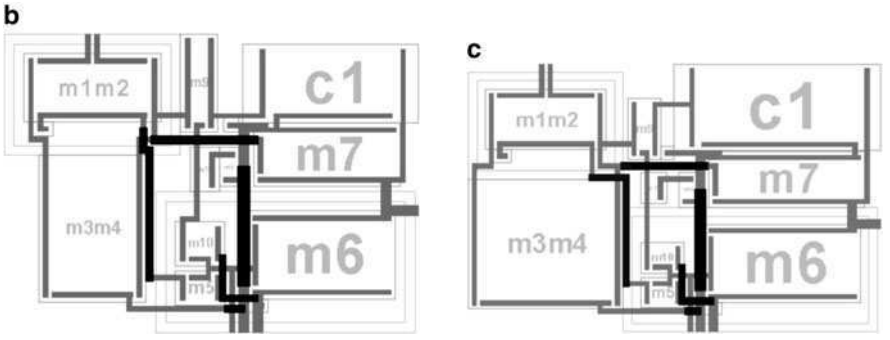
bottom(W2) = **top**(W1)
left(W2) = **left**(W1)
right(W2) = **right**(W3)
left(W3) = **left**(W4)
bottom(W3) = **bottom**(W2)
top(W3) = **bottom**(W4)
right(W2) - **left**(W2) >= **minWidth**(MET1)
right(W3) - **left**(W3) >= **minWidth**(MET1)
top(W2) - **bottom**(W2) = **minWidth**(MET1)
top(W3) - **bottom**(W3) = **minWidth**(MET1)

LDS of (b)

Fig. 4.24 Coding paths in LDS: path in (**a**) and the dynamic path in (**b**) are described in LDS in (**e**) and (**f**), respectively; orientation of a dynamic path may change as depicted in (**c**) and (**d**)

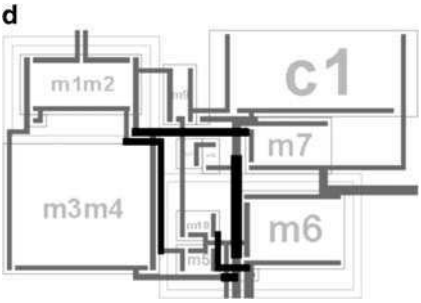


Two Stage Compansated OPAMP



Synthesized Layout (LDS is compiled.)

M3-M4 pair is enlarged



M6 and M7 are narrowed

Fig. 4.25 LDS code for the OPAMP of (**a**) is used to synthesize the layouts in (**b**), (**c**), and (**d**)

4.5.7 Other Routing Strategies

One method utilized in digital routing to overcome the wire congestion problems and address the net ordering effects has been probabilistic routing. Although the number of connections in analog routing is much fewer, this approach can be generalized to include analog routing constraints as well. [83] proposes the assignment of global routes to routing areas between modules. Then, a resistor array is created where each resistor represents a possible routing area for a net. The value of the resistor is assigned according to the probability of routing in that channel which depends not only on geometrical information, but also on some constraint information. The resistor array is then simulated and resistors corresponding to low probabilities are removed iteratively from the network, updating the probability values at each iteration, thus completing the assignment. [84] does not use a resistor array, but a heuristic, where the probabilities are converted to priorities for routing. Furthermore, a probability is assigned to each grid point rather than to a routing area. However, the probability formulation is quite detailed and calculates analog constraints including symmetry.

The analog router described in [85] implements x–y routing that utilizes an efficient scheme for generating candidate routes for each net. A single-layer routing option may also be used. These candidate routes are then simultaneously considered for compatibility, and finally a set of compatible routes is chosen. This routing method has been used before for multichip modules and was also extended to digital routing. In x–y routing, two adjacent metal layers are routed simultaneously. All horizontal segments are routed on one layer, with all vertical ones on the other. Adjacent segments on different layers are connected through stacked vias. Routing in each layer pair is done in phases, each phase generating candidate routes with different numbers of vias. The number of phases is at the designer's discretion, as are the types of candidate routes the designer wishes to generate. The algorithm discussed in [85] assumes all nets are two-terminal nets. But multiterminal nets may also be successfully routed. For any t-terminal net, a minimal spanning tree is determined consisting of $t-1$ edges. Each edge is considered as a two-terminal net during routing. Most of the candidate routes are generated within the bounding box of the net. The bounding box is the smallest rectilinear region containing all terminals of the net. Once the candidate routes are generated in a particular phase, a compatibility graph $G(V, E)$ is constructed. Each vertex in the graph represents a candidate route, and an edge placed between two candidate routes signifies that the two routes are incompatible and cannot be selected together in the final routing solution. Once the graph is constructed, it is reduced such that no edges remain in the graph. The resulting graph called the reduced compatibility graph (RCG) represents the set of vertices (or candidate routes) that are compatible with each other. A candidate may be chosen from the RCG for each net under consideration. The constraints on the parasitics and on the layout geometry and symmetry are a part of user specifications and apply to the nets specified. Various other constraints such as specifying wire widths, confining certain nets to specific layers, defining keep- out areas, specifying parallel distances, and coupled lengths between two segments as a function of their

widths, etc., can also be included by the designer. These are accounted for during the routing process to obtain optimized routing solutions respecting given constraints. This routing methodology based on candidate generation has an inherent global approach and tries to satisfy all the constraints simultaneously for all nets considered together instead of using an incremental net-by-net approach, which may be unable to route all nets respecting all the constraints as routing congestion increases. It was also shown that x–y routing can be extended to 45° routing as well [86].

4.6 Specialized Analog Routers

There are many applications in analog design, which may require entirely different or enhanced routers compared to those discussed above. Two such applications, namely, RF circuits and analog arrays are briefly illustrated below.

4.6.1 Routing for RF Circuits

The design of RF circuits has also been addressed recently in many design automation systems for analog integrated circuits. However, specific layout generation technologies in this respect have been rather few. CYCLONE [87,88] is a tool for the automatic design of VCO circuits. It proposes special module generation techniques, but the placement and routing are done by LAYLA. It is thus implicitly assumed that no special routing approaches are necessary for routing RF circuits.

On the other hand, CORAL [89] is a routing tool implicitly for RF circuits, and it adopts area routing because of the importance of routing parasitics. The A* algorithm is used for maze routing. Parasitic effects such as inductive and capacitive crosstalk are modeled in terms of the degradation induced on the characteristic impedance Z_0 and loss. Alternatively, at low frequencies discrete (R,C,L) parasitics can be used. Analytical models of all considered parasitics are obtained by fitting appropriate mathematical expressions to data obtained from 2-D or 3-D field solvers. The routing is performed in two phases. The first phase is constraint-based routing as described earlier. Layout synthesis of RF and microwave circuits almost always requires that the dimensions of some interconnect lines be fixed. However, length constraints on interconnect cannot be effectively enforced during this phase. Hence, the routing or constructive phase is followed by a refinement phase. The refinement consists of progressive expansion of all nets simultaneously thus allowing enforcement of all net constraints, while no new violations are created on the remaining parasitic constraints.

4.6.2 Routing for Analog Arrays

An important class of layout generation tasks has not been discussed yet. Frequently, in analog blocks a highly regular architecture of basic cells is used. Examples of this are flash type A/D converters, Cellular Neural Networks, or current-steering D/A converters. Typically, the regular layout structures used in analog blocks contain an array of unit cells (potentially with slightly different versions), which process one or more input signals in a parallel way and steer one or more output signals. Although routing automation for analog arrays was first mentioned in [90], this approach does not go far beyond maze routing with genetic optimization.

A real router for array style analog design is Mondriaan [91, 92]. In Mondriaan, the connections in and out of the array or internal to the array are realized through routing channels across or between the cells. The connectivity for ground, biasing or power supply connections is easily realized through abutment. Cells can be flipped upside down or sideways to share lines by abutment. Thus, the offered placement and routing functionality is much more powerful than the stretch and tile approach and covers the requirements of a large variety of analog circuits. Note that the placement and routing of field programmable gate arrays (FPGA) somewhat resembles this approach. An essential difference is that in FPGA routing, the majority of the connections are internal to the logic array, while in analog applications the majority of the connections are to pins at the edge of the matrix. Furthermore, the placement and routing of FPGAs are faced with a fixed number of wires and blocks and the critical delay (caused by routing) is to be minimized. This is not the case in analog applications: the number of wires is variable, but should be minimized, and the performance depends on equal capacitance, resistance or matching rather than the critical delay. Mondriaan generates a symbolic placement and routing from the floorplan, netlist, and symbolic basic cell. To accomplish this, a search algorithm is used to propagate the placement and connectivity information across the array. First, the number of vertical wires for every column is determined (if not specified by the user). Next, for all fixed IO pins free vertical wires are selected and the net of the wire is updated. If the cell connected to this net is not placed, it is placed in a free array slot. When all fixed IO pins have been connected, and all cells have been placed, the cells can be scanned columnwise, to propagate their connectivity. As a last step, the number of horizontal wires is determined (if it is not given by the user). This is done by counting the number of vertical wires, which have to be connected. Then the wires are scanned and free horizontal wires are selected to connect the vertical wires. Of course if no vertical wires need to be connected, no horizontal wires are created. The tool also contains special bus and tree generators.

The routing problem for field programmable analog arrays (FPAA) was mentioned as early as 1999 in [93]. The router FAAR makes connections between cells on a local and global level. The routing scenario is different from Mondriaan in that the number of switches is limited, thereby limiting the possible number of connections. The main properties of FPAA routers are as follows: The analog routers discussed so far are specifically targeted for full-custom designs. FPAA routing is more combinatorial in nature, and hence work needs to be done to extend ASIC

routing heuristics for FPAAs. More routing constraints will have to be incorporated since routing resources are fixed in number and preplaced, and there are constraints on permissible connections. Classical analog channel routing algorithms are not very suitable for array-based FPAAs because of the difficult nature of subdividing the routing problem into independent channels. FPGA routers cannot be used in their current form since they are targeted for routing on typical FPGA architectures. The FPAA routing considerations are different from the typical FPGA routing considerations. Graph-based FPGA routers use minimum-rectilinear-Steiner-tree (MRST) heuristics; this is not quite necessary in most FPAA cases because of the inability to have bends in a route owing to the single-segment architecture. Some FPGA routers can handle nonsegmented FPGA architectures or FPGA architectures with various types of segmentation distributions, and different switch box architectures. However, they do not describe completely the target FPAA routing architecture. Several modifications need to be made to handle a typical connection within switchbox architectures, as well as single-segment routing architecture. The issue of performance degradation, which is critical in FPAA routing is not addressed by any FPGA router. FAAR accepts as input a netlist of placed CABs (Computational Analog Block) and IO cells, and the parasitic bound for each net that limits the number of switches used. This bound keeps the performance degradation within acceptable limits. By modifying the architectural parameters including size of the CAB array, number of tracks per horizontal channel and vertical channel, FAAR may be used to route for array-based architectures. The following four subproblems can be identified to simplify the explanation of the routing problem in FPAAs:

1. Routing between two terminals of a net: The problem is to find the shortest path between two terminals of a net where the distance is defined by the number of routing resources required. There are typically four possible alternative routes for a source and destination terminal pair using the allowed connections:
   a. Using the local interconnect to make a connection between the terminals without utilizing any global wires.
   b. Using one global wire if it is an allowed connection for both terminals.
   c. Using two global wires (one horizontal and one vertical) if the terminals cannot share the same wire.
   d. Using three global wires, two horizontal and one vertical (or vice versa) if neither terminal can make a connection to its allowed horizontal (or vertical) wire because it was already used by another net.
2. Routing multiterminal nets: The problem is to find a minimum-length route between all terminals of a given net. Each net has one source terminal (output terminal of a CAB or IO cell) and multiple destination terminals (input terminals of CABs or IO cells). A multiterminal net can be viewed as a set of two-terminal nets, one two-terminal net between the source terminal and each of the destination terminals, and these two-terminal nets can be routed sequentially. All local connections are completed first and then the unconnected terminals are routed using global wires. Given a partial multiterminal net-route, sharing should be maximized and as few additional routing resources as possible should be used.

3. Routing multiple nets: The problem here is to find satisfactory routes for several nets simultaneously. This is the compatibility problem, and it is NP-complete. However, FAAR uses sequential routing and this problem is not addressed.
4. Performance constrained FPAA routing: The main performance degrading parasitics are identified as the number of switches (which bring additional resistance and capacitance to the corresponding net) and the number of net crossings (which bring crosstalk). After each net is routed, its parasitic is checked for violation of the bound. If the net's parasitic bound is violated, the net fails to keep its performance degradation within acceptable limits, and the net is ripped-up and the routing is re-tried.

When routing a net, FAAR initially makes all possible connections using local interconnect. FAAR then sequentially routes each unrouted destination terminal to the source terminal using global interconnect. Each two-terminal route is added to the existing partial net-route. As it builds the net-route, FAAR tries to maximize reuse of the routing resources in the partial net-route. Every routing resource used by a net is then blocked from future use.

To achieve high-performance, channel segmentation and buffer insertion are proposed in [94]. These are actually ideas borrowed from FPGA design. Also, the combined application of buffer insertion and segmentation will yield more optimal results in terms of delay matching. Another interesting suggestion for better area usage, and thus more optimal routing is hexagonal structures and reconfigurable CABs [95]. The CABs in this example are all digitally configurable gm-C filters. Thus, every CAB can be configured on location such that the necessary interconnections are minimized.

A more recent approach [96], on the other hand, uses a simulator in the loop approach, where the effects of the interconnect are included in the CAB simulation and the CAB is configured accordingly. In the extreme case, the parasitics of the interconnect can even be useful as they can form part of a filter.

4.7 Manufacturability and Yield Issues in Routing

The goal of a performance-driven routing tool is to route an analog circuit such that the performance degradation caused by layout parasitics remains within the specification margins imposed by the designer. For a given set of circuit specifications, several valid routing solutions can be found. Among these, the choice should be toward those solutions with higher yield and easier manufacturability and testability. Several algorithms have been proposed to increase yield in routing. However, these have been only for digital channels. One of the first studies oriented toward yield maximization in analog routing is [97]. Sensitivity analysis and line expansion routing are at the core of the performance-driven router. If the performance-driven routing phase is successful and there is enough performance margin left, a yield and testability optimization loop is entered. During this loop, nets are removed and rerouted until the available performance margin is consumed or no further yield/testability improvement is found. During rerouting of a net, the geometry of all

other nets is known. Therefore, the additional performance degradation introduced by a partial path can be computed exactly, and if it exceeds the available performance margin, the partial path can be removed from the search heap. During the search, the expected number of bridging faults is calculated and these are added to the cost, thus ensuring that these will be minimized while searching for the best route. The bridging faults modeled are of two categories; dielectric pinholes and photolithographic defects. Dielectric pinholes are defects, which often occur in chip insulators. Their occurrence can result in a short between wires at different routing levels. The critical area associated with these defects is the overlap region between two wires. Photolithographic defects can cause shorts between wires on the same routing level. The expected number of faults for two parallel conductors, separated by a narrow slit can be calculated by combining the critical area in function of defect size and a defect size distribution. This approach will result in higher yield, but this measure is incapable of distinguishing between testable designs. Power supply current monitoring technique was assumed to be the testing methodology for [97]. To this end, the distribution currents of the 'good' circuit were obtained via Monte Carlo simulation. All faulty circuits were also simulated and their currents were obtained for all cases and the configuration with the highest separation was selected.

The problem of electromigration was addressed by developing a current-driven router in [98, 99] and further extended in [100] and [101]. The current in each path was determined by simulation, either based on input patterns provided by the user, or based on Monte Carlo type simulation. The wire widths were determined accordingly. Connection of multiterminal nets is done in a Steiner-tree fashion. However, a basic Steiner tree approach is not enough since the wire widths may be different for each section of the wire. A greedy method was utilized in which a Steiner point is constructed from three terminals. The currents from two terminals are also summed at this Steiner point and the width of the wire segment is calculated. A new Steiner point and a new current is found by combining the previous point with the next two terminals to be routed. The algorithm continues in this fashion until the whole tree is constructed. Finally, a current density simulator was developed to verify the layout. A complementary approach would be based on a terminal tree, which defines a detailed terminal-to-terminal routing sequence with known terminal currents. A current-driven detailed router must solve the problem of altering current strengths in a prior routed subnet whenever a new terminal is linked to it. To allow for a current calculation based on Kirchhoff's current laws prior to detailed routing, at least the sequence of all terminals to be connected must be known. Added detailed routing connections which directly link a new (not yet connected) terminal with its respective target terminal will then have no influence on current strengths calculated in the prior routed subnet (with the calculation based on all terminals). Hence, the most coarse grain approach possible for current-driven routing without postrouting layout modification is based on a predefined terminal-to-terminal routing sequence. The integration of this methodology within a commercial layout tool was also demonstrated.

The above algorithms calculate the wire widths after constructing a terminal tree that obtains a minimum total length, and later minimize the total area of the

routing net. Another approach would be to construct the terminal tree considering area minimization first [102]. Then, simulated annealing can be applied, using the terminal tree obtained above as the initial solution. The terminal tree will be constructed in a bottom-up manner from the leaf nodes to the root node. At each stage, the terminal with minimum current value is selected, and its nearest neighbor is found. Then, the pair of terminals is added into the terminal tree. While generating a terminal pair, the width of the wire between the two terminals is determined. The last terminal added into the terminal tree will be the root node.

4.8 Conclusions

This chapter has given an overview of routing techniques for analog layout synthesis. Routers for full-custom analog circuits have been discussed based on the cost functions they are trying to minimize. Furthermore, various data representation strategies have been presented, and their suitability for routing has been explored. As a second routing problem, template-based routing has been dealt with. A new approach for this routing problem has been presented. Finally, routers for RF circuits and analog arrays have been discussed in addition to manufacturing and yield issues.

In our opinion, the problem of routing analog circuits is still an open problem. As mentioned in the very beginning of this chapter, the performance of routing is directly affected by previous layout steps, such as placement, partitioning, and module generation. One research direction would be to put more effort into one-step layout generation rather than the conventional sequential approach. Another open problem is the routing of RF circuits, where every interconnect is actually a device. These interconnects must be carefully designed and modeled in the final layout. As the technology moves to deeper submicron dimensions, design rules get more complicated. Furthermore, manufacturability and yield of a circuit become very important issues. Simple improvements over well-known routing approaches will not be enough to perform analog routing in such advanced technologies. New algorithms will have to be developed. Finally, template-based layout generation, however primitive it may seem as an idea, has many applications and will probably become a commonly used layout generation approach in the near future. The same problems, namely, routing at RF frequencies, or yield aware routing will still be valid for template-based approaches as well.

References

1. C.Y. Lee. An algorithm for path connection and its applications. *Electronic Computers, IRE Transactions on*, EC-10:346–365, 1961
2. E.F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292, 1959

3. S. Akers. *Design Automation of Digital Systems: Theory and Techniques*, volume 1, chapter 6. Prentice-Hall, NJ, 1972
4. S. Akers. A modification of Lee's path connection algorithm. *Electronic Computers, IEEE Transactions on*, EC-16(2):97–98, 1967
5. F.O. Hadlock. A shortest path algorithm for grid graphs. *Networks*, 7(4):323–334, 1977
6. P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum paths in graphs. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968
7. J. Soukup. Fast maze router. In *Proceedings of Design Automation Conference*, pages 100–102, 1978
8. K. Mikami and K. Tabuchi. A computer program for optimal routing of printed circuit connectors. In *IFIPS Proceedings*, volume H47, pages 1475–1478, 1968
9. D.W. Hightower. A solution to line routing problems on the continuous plane. In *Proceedings of Design Automation Conference*, pages 1–24, 1969
10. C.J. Alpert, D.P. Mehta, and S.S. Sapatnekar. Handbook of algorithms for physical design automation. CRC Press, 2009
11. A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of Design Automation Conference*, pages 155–169, 1971
12. D.N. Deutsch. A "Dogleg" channel router. In *Proceedings of Design Automation Conference*, pages 425–433, 1976
13. B.W. Kernighan, D.G. Schweikert, and G. Persky. An optimum channel-routing algorithm for polycell layouts of integrated circuits. In *Proceedings of Design Automation Conference*, pages 57–66, 1988
14. R.L. Rivest and C.M. Fiduccia. A "GGreedy" channel router. In *Proceedings of Design Automation Conference*, pages 256–262, 1988
15. S. Gueron and R. Tessler. The fermat-steiner problem. *The American Mathematical Monthly*, 109:443–451, 2002
16. F.K. Hwang. On steiner minimal trees with rectilinear distance. *SIAM J. Appl. Math.*, 30: 37–58, 1976
17. S. Futagami, I. Shirakawa and H. Ozaki. An automatic routing system for single-layer printed wiring boards. *Circuits and Systems, IEEE Transactions on*, CAS-29(1):46–51, 1982
18. International Symposium on Physical Design 2007. http://www.sigda.org/ispd2007/rcontest/
19. E. Malavasi and A. Sangiovanni-Vincentelli. Area routing for analog layout. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 12(8):1186–1197, Aug 1993
20. J.K. Ousterhout. Corner stitching: A data-structuring technique for VLSI layout tools. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 3(1): 87–100, 1984
21. J.M. Cohn, D.J. Garrod, R.A. Rutenbar, and L.R. Carley. KOAN/ANAGRAM II: New tools for device-level analog placement and routing. *Solid-State Circuits, IEEE Journal of*, 26(3):330–342, 1991
22. Z. Xing and R. Kao. Shortest path search using tiles and piecewise linear cost propagation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(2):145–158, 2002
23. R.H.J.M. Otten. Automatic floorplan design. In *Proceedings of Design Automation Conference*, pages 261–267, 1982
24. H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajitani. VLSI module placement based on rectangle-packing by the Sequence-Pair. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(12):1518–1524, 1996
25. P.N. Guo, C.K. Cheng, and T. Yoshimura. An O-Tree representation of non-slicing floorplan and its applications. In *Proceedings of Design Automation Conference*, pages 268–273, 1999
26. N. Fu, S. Nakatake, and M. Mineshima. Multi-SP: A representation with united rectangles for analog placement and routing. In *Proceedings of IEEE Computer Society Annual Symposium*, page 6, 2006

27. S. Nakatake, K. Sakanushi, Y. Kajitani, and M. Kawakita. The channeled-BSG: A universal floorplan for simultaneous place/route with IC applications. In *Proceedings of the International Conference on Computer-Aided Design*, pages 418–425, 1998
28. M. Mogaki, Y. Shiraishi, M. Kimura, and T. Hino. Cooperative approach to a practical analog LSI layout system. In *Proceedings of the Design Automation Conference*, pages 544–549, 1993
29. D.J. Chen and B.J. Sheu. Generalised approach to automatic custom layout of analogue ICS. *IEE Proceedings of G Circuits, Devices and Systems*, pages 481–490, 1992
30. E. Malavasi, M. Chilanti, and R. Guerrieri. A general router for analog layout. In *Proceedings of VLSI and Computer Peripherals*, pages 5/49–5/51, 1989
31. E. Malavasi, E. Charbon, E. Felt, and A. Sangiovanni-Vincentelli. Automation of IC layout with analog constraints. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(8):923–942, 1996
32. W.H. Kao, C.Y. Lo, M. Basel, and R. Singh. Parasitic Extraction: Current state of the art and future trends. In *Proceedings of the IEEE*, pages 729–739, May 2001
33. J.A. Davis and J.D. Meindl. Compact distributed RLC interconnect models-part II: Coupled line transient expressions and peak crosstalk in multilevel networks. *Electron Devices, IEEE Transactions on*, 47(11):2078–2087, Nov 2000
34. M. Mogaki, N. Kato, Y. Chikami, N. Yamada, and Y. Kobayashi. LADIES: An automatic layout system for analog LSI's. In *Proceedings of International Conference on Computer-Aided Design*, pages 450–453, 1989
35. H. Shin and A. Sangiovanni-Vincentelli. Mighty: A "rip-up and reroute" detailed router. In *Proceedings of International Conference on Computer-Aided Design*, pages 10–13, 1986
36. C.H. Séquin, H.Y. Koh and P.R. Gray. Automatic layout generation for CMOS operational amplifiers. In *Proceedings of International Conference on Computer-Aided Design*, pages 548–551, 1988
37. J. Litsios, J. Rijmenants, T. Schwarz and R. Zinszner. ILAC: An automated layout tool for analog CMOS circuits. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 7.6/1–7.6/4, 1988
38. T. Schwarz, J. Rijmenants, J. Litsios and M.G.R. Degrauwe. ILAC: An automated layout tool for analog CMOS circuits. *Solid State Circuits, IEEE Journal of*, 24(12):417–425, 1989
39. M. Declercq, M. Kayal, S. Piguet and B. Hochet. An interactive layout generation tool for CMOS analog ICS. In *Proceedings of International Symposium on Circuits and Systems*, volume 3, pages 2431–2434, 1988
40. M. Declercq, M. Kayal, S. Piguet and B. Hochet. Salim: A layout generation tool for analog ICS. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 7.5/1–7.5/4, 1988
41. S. Piguet, F. Rahali, M. Kayal, E. Zysman, and M. Declercq. A new routing method for full custom analog ICS. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 27.7/1–27.7/4, 1990
42. J.C. Lee, D.J. Chen and B.J. Sheu. Slam: A smart analog module layout generator for mixed analog-digital VLSI design. In *Proceedings of International Conference on Computer Design*, pages 24–27, 1989
43. S.M. Gowda J.C. Lee and B.J. Sheu. Fully automated layout generators for high-performance analog VLSI modules. In *Proceedings of IEEE Region 10 International Conference*, pages 893–896, 1989
44. Z.M. Lin. Global routing techniques for an automatic mixed analog/digital IC layout compiler. In *IEEE Proceedings of Southeastcon*, volume 1, pages 392–396, 1991
45. M.F. Chowdhury and R.E. Massara. An expert system for general purpose analogue layout synthesis. In *Proceedings of Midwest Symposium on Circuits and Systems*, volume 2, pages 1171–1174, 1990
46. M.F. Chowdhury, R.E. Massara, and H. Tang. Analogue layout synthesis based on a planning scheme using artificial intelligence. In *Proceedings of IEEE International Sympoisum on Circuits and Systems*, volume 5, pages 3094–3097, Jun 1991

47. D.J. Garrod, R.A. Rutenbar, and L.R. Carley. Automatic layout of custom analog cells in ANAGRAM. In *Proceedings of International Conference on Computer-Aided Design*, pages 544–547, 1988
48. R.L. Rivest, T.H. Cormen and C.E. Leiserson. *Introduction to Algorithms*. MIT, MA, 1990
49. B. Basaran, R.A. Rutenbar, and L.R. Carley. Latchup-aware placement and parasitic-bounded routing of custom analog cells. In *Proceedings of International Conference on Computer-Aided Design*, pages 415–421, 1993
50. U. Choudhury and A. Sangiovanni-Vincentelli. Constraint generation for routing analog circuits. In *Proceedings of Design Automation Conference*, pages 561–566, 1990
51. U. Choudhury and A. Sangiovanni-Vincentelli. Use of performance sensitivities in routing analog circuits. In *Proceedings of IEEE International Symposium on Circuits and Systems*, volume 1, pages 348–351, 1990
52. U. Choudhury and A. Sangiovanni-Vincentelli. Automatic generation of parasitic constraints for performance-constrained physical design of analog circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 12(2):208–224, 1993
53. U. Choudhury and A. Sangiovanni-Vincentelli. Constraint-based channel routing for analog and mixed analog/digital circuits. In *Proceedings of International Conference on Computer-Aided Design*, pages 198–201, 1990
54. U. Choudhury and A. Sangiovanni-Vincentelli. Constraint-based channel routing for analog and mixed analog/digital circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 12(4):497–510, 1993
55. G. Gad-El-Karim and R.S. Gyurcsik. Generation of performance sensitivities for analog cell layout. In *Proceedings of the Design Automation Conference*, pages 500–505, 1991
56. E. Malavasi, U. Choudhury, and A. Sangiovanni-Vincentelli. A routing methodology for analog integrated circuits. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 202–205, 1990
57. N. Nilsson. *Problem-Solving Methods in Artificial Intelligence*. McGraw-Hill, New York, 1971
58. G.W. Clow. A global routing algorithm for general cells. In *Proceedings of the Design Automation Conference*, pages 45–51, 1984
59. S. Prasitjutrakul and W.J. Kubitz. A timing-driven global router for custom chip design. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 48–51, 1990
60. R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1(4):290–306, 1972
61. J.M. Cohn, D.J. Garrod, R.A. Rutenbar, and L.R. Carley. Techniques for simultaneous placement and routing of custom analog cells in KOAN/ANAGRAM II. In *Proceedings of IEEE International Conference on Computer-Aided Design*, pages 394–397, 1991
62. N.U.D. Gohar, P.Y.K. Cheung, and C.K. Pun. Rachana: An integrated placement and routing approach to CMOS analog cells. In *Proceedings of IEEE International Symposium on Circuits and Systems*, volume 6, pages 2981–2984, 1992
63. J.A. Prieto, J.M. Quintana, A. Rueda, and J.L. Huertas. An algorithm for the place-and-route problem in the layout of analog circuits. In *Proceedings of IEEE International Symposium on Circuits and Systems*, volume 1, pages 491–494, 1994
64. J.A. Prieto, A. Rueda, J.M. Quintana, and J.L. Huertas. A performance-driven placement algorithm with simultaneous place&route optimization for analog IC's. *Proceedings of European Design and Test Conference*, pages 389–394, 1997
65. Y. Kubo, S. Nakatake, Y. Kajitani, and M. Kawakita. Explicit expression and simultaneous optimization of placement and routing for analog IC layouts. In *Proceedings of Asia and South Pacific Design Automation Conference*, page 467, 2002
66. L. Zhang and Y. Jiang. Global-routing driven placement strategy in analog VLSI physical designs. In *Proceedings of IEEE International Symposium on Circuits and Systems*, volume 2, pages 1239–1242, 2005

67. H. Zhang, P. Karthik, H. Tang, and A. Doboli. An explorative tile-based technique for automated constraint transformation, placement and routing of high frequency analog filters. In *Proceedings of IEEE International Symposium on Circuits and Systems*, volume 6, pages 5629–5632, 2005
68. L. Zhang and U. Kleine. A novel analog layout synthesis tool. In *Proceedings of the International Symposium on Circuits and Systems*, volume 5, pages V-101–V-104, 2004
69. L. Zhang, U. Kleine, and Y. Jiang. An automated design tool for analog layouts. *Very Large Scale Integration Systems, IEEE Transactions on*, 14(8):881–894, Aug 2006
70. T.E. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms (Second ed.)*, section 24.3. MIT, MA, 2001
71. U. Kleine L. Zhang and M. Wolf. Automatic inner wiring for integrated analog modules. In *Proceedings of Mixed Design of Integrated Circuits and Systems*, pages 109–114, 2001
72. L. Schreiner, M. Olbrich, E. Barke, and V. Meyer zu Bexten. Parsy: A parasitic symmetric router for net bundles using module generators. In *International Symposium on VLSI Design, Automation and Test*, pages 71–74, 2005
73. E. Yılmaz and G. Dündar. New layout generator for analog CMOS circuits. In *European Conference on Circuit Theory and Design*, pages 36–39, 2007
74. E. Yılmaz and G. Dündar. Analog layout generator for CMOS circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 28(1):32–45, Jan 2009
75. J.D. Conway and G.G. Schrooten. An automatic layout generator for analog circuits. In *Proceedings of European Conference on Design Automation*, pages 513–519, 1992
76. B.R. Owen, R. Duncan, S. Jantzi, C. Ouslis, S. Rezania, and K. Martin. BALLISTIC: An analog layout language. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 41–44, 1995
77. M. Dessouky and M.M. Louërat. A layout approach for electrical and physical design integration of high-performance analog circuits. *Proceedings of International Symposium on Quality Electronic Design*, pages 291–298, 2000
78. M. Dessouky, M.M. Louërat, and J. Porte. Layout-oriented synthesis of high performance analog circuits. In *Proceedings of Conference on Design, Automation and Test in Europe*, pages 53–57, 2000
79. R. Castro-Lopez, F.V. Fernandez, M. Delgado-Restituto, F. Medeiro, and A. Rodriguez-Vazquez. Creating flexible analogue IP blocks. In *Proceedings of Solid-State Circuits Conference*, pages 437–440, 2001
80. R. Castro-Lopez, O. Guerra, E. Roca, and F.V. Fernandez. An integrated layout-synthesis approach for analog ICS. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(7):1179–1189, 2008
81. N. Lourenco and N. Horta. LAYGEN - An evolutionary approach to automatic analog IC layout generation. In *Proceedings of IEEE International Conference on Electronics, Circuits and Systems*, pages 1–4, 2005
82. N. Lourengo, M. Vianello, J. Guilherme, and N. Horta. LAYGEN - Automatic layout generation of analog ICS from hierarchical template descriptions. In *Research in Microelectronics and Electronics*, pages 213–216, 2006
83. K. Okada, H. Onodea, and K. Tamaru. A global routing algorithm for analog circuits using a resistor array model. In *Proceedings of IEEE International Symposium on Circuits and Systems*, volume 4, pages 667–670, 1996
84. C. Du, Y. Cai, and X. Hong. A performance driven probabilistic resource allocation algorithm for analog routers. In *Midwest Symposium on Circuits and Systems*, pages 730–733, 2008
85. K. Sajid, J.D. Carothers, J.J. Rodriguez, and W.T. Holman. Global routing methodology for analog and mixed-signal layout. In *Proceedings of IEEE International ASIC/SOC Conference*, pages 442–446, 2001
86. S. Kumar, J.D. Carothers, R.D. Newbould, and B.V. Krishnan. Candidate generation for 45 degree routing for mixed-signal layout. In *Southwest Symposium on Mixed-Signal Design*, pages 233–236, 2003

87. M. Steyaert .G. Gielen C. De Ranter, G. Van der Plas and W. Sansen. CYCLONE: Automated design and layout of RF LC-oscillators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(10):1161–1170, 2002
88. C. De Ranter, B. De Muer, G. Van der Plas, P. Vancorenland, M. Steyaert, G. Gielen, and W. Sansen. CYCLONE: Automated design and layout of RF LC-oscillators. In *Proceedings of the Design Automation Conference*, pages 11–14, 2000
89. B. Donecker E. Charbon, G. Holmlund and A. Sangiovanni-Vincentelli. A performance-driven router for RF and microwave analog circuit design. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 383–386, 1995
90. H.G. Wolf and D.A. Mlynski. A new genetic single-layer routing algorithm for analog transistor arrays. In *Proceedings of IEEE International Symposium on Circuits and Systems*, volume 4, pages 655–658, 1996
91. G. Van der Plas, J. Vandenbussche, G. Gielen, and W. Sansen. Mondriaan: A tool for automated layout synthesis of array-type analog blocks. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 485–488, 1998
92. G. Van der Plas, J. Vandenbussche, G.G.E. Gielen, and W. Sansen. A layout synthesis methodology for array-type analog blocks. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 21(6):645–661, 2002
93. S. Granesan and R. Vemuri. FAAR: A router for field-programmable analog arrays. In *Proceedings of International Conference on VLSI Design*, pages 556–563, Jan 1999
94. H. Huang, J.B. Bernstein, M. Peckerar, and Ji Luo. Combined channel segmentation and buffer insertion for routability and performance improvement of field programmable analog arrays. In *Proceedings of IEEE International Conference on Computer Design*, pages 490–495, 2004
95. J. Becker and Y. Manoli. A new architecture of field programmable analog arrays for reconfigurable instantiation of continuous-time filters. In *Proceedings of IEEE International Conference on Field-Programmable Technology*, pages 367–370, 2004
96. F. Baskaya, D.V. Anderson, and Sung Kyu Lim. Net-sensitivity-based optimization of large-scale field-programmable analog array (FPAA) placement and routing. *Circuits and Systems II: Express Briefs, IEEE Transactions on*, 56(7):565–569, 2009
97. K. Lampaert, G. Gielen, and W. Sansen. Analog routing for manufacturability. In *Proceedings of IEEE Custom Integrated Circuits Conference*, pages 175–178, 1996
98. T. Adler, H. Brocke, L. Hedrich, and F. Barke. A current driven routing and verification methodology for analog applications. In *Proceedings of Design Automation Conference*, pages 385–389, 2000
99. T. Adler and E. Barke. Single step current driven routing of multiterminal signal nets for analog applications. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pages 446–450, 2000
100. J. Lienig, G. Jerke, and T. Adler. Electromigration avoidance in analog circuits: two methodologies for current-driven routing. In *Proceedings of Asia and South Pacific International Conference on VLSI Design Automation Conference*, pages 372–378, 2002
101. J. Lienig and G. Jerke. Current-driven wire planning for electromigration avoidance in analog circuits. In *Proceedings of Asia and South Pacific International Conference on VLSI Design Automation Conference*, pages 783–788, 2003
102. B. Xue and X. He. Electromigration avoidance aware net splitting algorithm in analog circuits. In *International Conference on Communications, Circuits and Systems Proceedings*, volume 4, pages 2805–2808, 2006

Part III
Layout in the Design Flow

Chapter 5
Analog Layout Retargeting

Hazem Said, Mohamed Dessouky, Reem El-Adawi, Hazem Abbas, and Hussein Shahein

Abstract This chapter focuses on analog layout process retargeting. Unlike automatic placement and routing tools, retargeting starts with an input layout in a given process. The main target is to conserve most of the layout physical intelligence while migrating it to another given technology. This is usually achieved by adapting existing layout compaction techniques borrowed from the digital world. Historically, layout compaction used to rely on fast constraint-graph operations. More recently, linear programming has been introduced to support hierarchy in addition to complex analog constraints. This chapter introduces a novel graph-based simplex algorithm that combines the efficiency of graph-based methods together with the generality of linear programming ones. It also allows symmetry, hierarchy, and cell replacement support to be integrated seamlessly without any artificial modification of the algorithm. For simple layout constraints, the algorithm complexity tends to be as linear as graph-based techniques, while for the most complex constraints and objective function it tends to that of the simplex method.

5.1 Introduction

Driven by market needs, semiconductor fabrication houses continue to enhance technologies toward smaller transistor feature sizes. This puts pressure on mixed-signal design teams. From one side, they have to come up with new circuit architectures that make use of such powerful technologies by pushing device characteristics to their limits. From the other side, they have to migrate their legacy in-house intellectual property blocks (IPs) to such new processes. Apart from the few state-of-the-art blocks that benefit from the enhanced transistor performance, a lot of designs are just retargeted to the new process without any major performance changes.

M. Dessouky (✉)
Ain Shams University, 1 El-Sarayat St. Abbasia, Cairo 11517, Egypt
and
Mentor Graphics, 78 El Nozha St., Heliopolis, Cairo 11361, Egypt
e-mail: Mohamed_Dessouky@mentor.com

H.E. Graeb (ed.), *Analog Layout Synthesis: A Survey of Topological Approaches*,
DOI 10.1007/978-1-4419-6932-3_5, © Springer Science+Business Media, LLC 2011

On the digital side, most of the migration effort is a setup one spent in migrating the digital cell library. Digital designers are not involved during such phase. Migrating most of digital designs is just a matter of re-running the fully-automated design flow scripts already verified on the original design. This is not the case on the analog side. Analog design reuse is still a mostly manual process. Due to device characteristics change, the circuit is redesigned each time a chip is migrated to a new technology, even to achieve the same performance of the original design. This is a very time-consuming and resource intensive task. In fact, a lot of effort is wasted in just retargeting these blocks to the new process without any major performance changes. In some cases, specially on the IC component level, design migration is driven by the sole fact that a process becomes obsolete, where all designs on such process must be migrated to a newer one. As a result, there is an increasing demand in redesigning functional mixed-signal designs for new processes.

New designs are mostly done manually by expert designers. Sometimes, the electrical and/or physical design of those modified architectures are performed using automatic optimization tools. However, the need for automation is even stronger in the process migration case, where blocks are required to retain source design specifications together with the corresponding layout placement and routing. While some analog designers still argue on the use of automatic layout tools, when it comes to design migration, where there is not much that *creativity* associated with a new design, designers seem to accept a large degree of automation.

This chapter focuses on automatic analog layout migration. It is shown that layout compaction retains most of the source layout characteristics and heuristics common in the analog world. Most of the chapter is devoted to different compaction algorithms, while introducing a newly presented generic graph-based one. Section 5.2 presents previous work done in this direction. Section 5.3 introduces a typical tool flow and shows where does the compaction engine fits. Section 5.4 discusses different compaction approaches while providing the necessary background for the generic algorithm introduced in Sect. 5.5. Section 5.6 stresses the importance of defining the right set of compaction constraints for layout migration. Section 5.7 rapidly goes through different practical issues that face any industrial tool for layout retargeting. Some migration examples are then given in Sect. 5.8. They vary from simple design cases to more complex industrial-level layouts. Finally, conclusions are drawn in Sect. 5.9.

5.2 Previous Work

The need to migrate hard IPs was first investigated for digital cells [1]. Later, more focus was reinforced on the special needs of digital library IPs, such as port matching, power/ground size, etc. [2]. Analog IPs impose additional constraints to the retargeting process, such as matching and symmetry [3]. In fact, the behavior of many analog circuits is closely related to the corresponding physical design [4]. Therefore, source layout contains valuable design knowledge, which is usually already verified through chip fabrication and testing.

Most designers have been reluctant to use optimization-based tools for retargeting. Such tools tend to reinvent the wheel and produce newly optimized designs [5]. Despite the fact that resulting layouts might even look better than source ones, designers prefer not to take the risk of a *new* design and prefer the original layout, which has already gone through the whole fabrication cycle. In most cases, designers want to keep all design choices and knowledge of the source design.

One of the first serious trials to migrate analog cells based on an original design was presented in [6]. The information reused from the original layout was the relative positions of the building blocks and their aspect ratios. All blocks part of the floorplan are then generated automatically. Nothing guarantees that the devices in the building blocks will resemble their counterparts in the source layout.

Later, inspired by the work initiated in the digital domain for compaction-based layout migration, analog-specific tools extended this work in [7, 8]. Layout migration by compaction keeps the same physical knowledge, i.e., floorplan, placement and routing, so precious to analog designers. This approach is most appealing since the target layout looks very close to the source one. A recent migration framework aiming to achieve this has been presented in [8]. It comprises both device resizing and layout migration. Both engines are based on a *design extraction* methodology, during which all relevant design data are extracted. The layout part of this framework is discussed in more detail in Sects. 5.3 and 5.5.

5.3 Analog Layout Retargeting Flow

A typical compaction-based layout migration flow is illustrated in Fig. 5.1. The dashed box contains the different modules of the retargeting tool. The input to the tool includes:

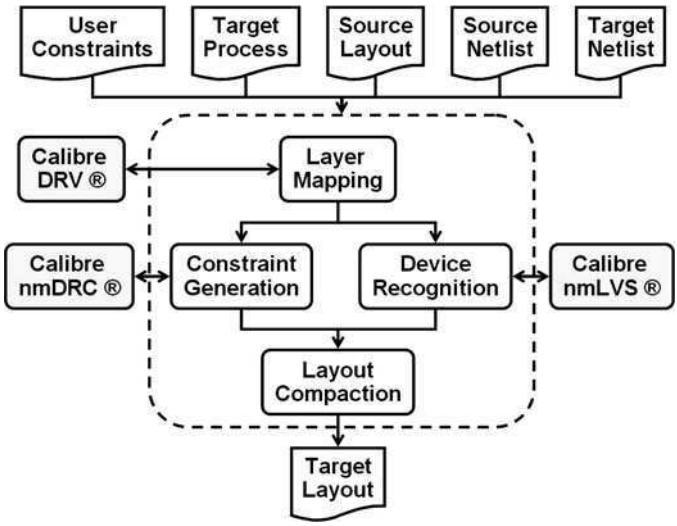


Fig. 5.1 Analog layout retargeting flow

- Source design netlist and layout.
- Target netlist with new device dimensions.
- Target process technology parameters and design rules.
- Optional user-defined constraints.

The flow includes three design extraction modules, namely: *Layer Mapping*, *Constraint Generation*, and *Device Recognition*, followed by the layout migration *Compaction* step. Each of these modules is briefly described in the following subsections. An important aspect of the flow is the multiple use of the industry-standard physical verification tool *Calibre*® [9]. This has the following advantages:

- Handling all process information files. This saves a lot of development time.
- New process file updates are already available and updated in the proper format.
- Taking advantage of the tool high performance and multitude of functionalities as will be shown below.

These makes the migration tool tightly coupled with the industrial one.

5.3.1 Layer Mapping

The layer mapping module copies all layout layers of the source process layout to the corresponding layers in the target process. This is done using the Calibre DESIGNrev™ [9] tool using a mapping file. This might involve mapping one layer to several ones or vice versa, depending on the source and target layer definitions. In addition, adjacent contacts and vias are merged to facilitate compaction, refer to Sect. 5.6. In some cases, the entire device needs to be completely replaced by another device in the target process due to the unavailability of a one-to-one correspondence, refer to Sect. 5.7.4. The mapping rule file is prepared once for each couple of source and target technologies. The output of this module is a layout that is similar to the source layout but in the target process layers.

5.3.2 Device Recognition

The target netlist input is generated by a separate netlist migration engine [10], where design performance is tuned in the target process to achieve the same source design performance. During such stage, most device sizes are often changed. In order that the migration tool applies such new device dimensions to the target layout, it should be able to recognize each device in the source layout and link it to the corresponding element in the netlist. A special device recognition module is thus needed. It involves the same operations as in a conventional *layout-versus-schematic* check tool usually used in a typical design flow. This is easily achieved using the Calibre nmLVS™ [9] tool using a special rule file. The tool is capable of identifying complex device structures such as finger and matched transistors and link them to a single device.

5.3.3 Constraint Generation

The source layout complies with the source process design rules. These must be translated to the corresponding target process rules and formulated in layout constraints that should be imposed on the target layout. This is not a one-to-one translation, since design rules change considerably from one process to the other. The Calibre nmDRC™ [9] tool is also employed in this module. The tool's main functionality is to check for design-rule errors in a given layout based on a rule file supplied by the foundry for each process. In a special internal mode of operation, it can also transform such rules to layout constraints by a modification of the rule file. This masks any kind of constraint complexity and employs the latest industrial rule check technology to generate them.

Constraint generation is critical to the compaction-based migration engine. The number of constraints has a huge impact on the efficiency and accuracy of compaction. It is important to reduce the constraints to the minimum possible set and remove any redundancy, refer to Sect. 5.6.

In addition to design-rule constraints, the updated geometrical device parameters generated by the netlist migration engine, e.g., transistor new length and width, are also converted to layout constraints on the physical device dimensions.

Layout migration is achieved by imposing both design-rule constraints and updated device dimension constraints on the layer-mapped layout using a layout compaction module. Details of the compaction engine are presented in Sect. 5.5.

5.4 Layout Compaction Methodologies: Background

The aim of compaction is to minimize the total layout area while satisfying all design-rule constraints of a specific process. Most compaction implementations in literature are based on various forms of constraint-graphs and linear programming [11, 12]. In this section, an introduction to these two approaches is elaborated showing strengths and weaknesses of each. This foundation is necessary to understand the compaction algorithm introduced in Sect. 5.5.

5.4.1 The Constraint-Graph Approach

The constraint-graph is a one-dimensional compaction technique, which uses a directed graph to capture design-rule constraints [13]. It remains one of the most popular approaches, thanks to its flexibility and efficiency [11].

During horizontal compaction, each element is represented by the x-coordinates of its edges. If the x-coordinates of all layout edges are indicated by the set: $x_1, x_2, \ldots, x_k$, a minimum-distance design-rule between two elements can now be expressed by an inequality of the form:

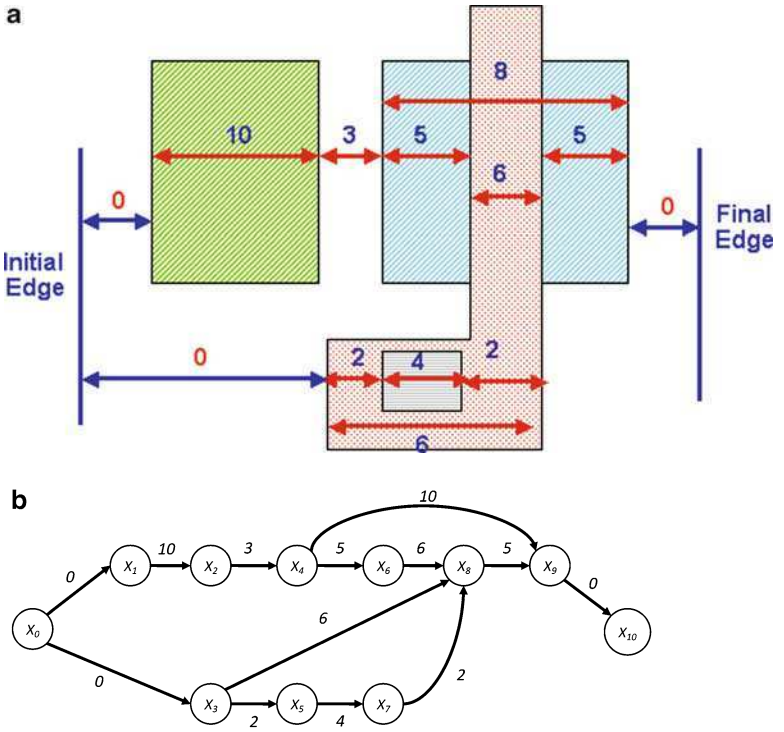


Fig. 5.2 (**a**) Sample layout with design-rule constraints and (**b**) the corresponding constraint-graph

$$x_j - x_i \geq d_{ij} \qquad \text{where } i, j \in (1 \ldots k), \tag{5.1}$$

where d_{ij} is the minimum distance spacing between elements located at x_j and x_i.

Consider the sample layout shown in Fig. 5.2a composed of a single transistor to the right of a single layout rectangle. Design rules are indicated by arrows. If all shown design rules are represented by inequalities of the form of (5.1), it is now possible to represent such inequalities in a so-called *constraint-graph*, $G(V, E)$, as follows:

- The vertex set, V, is constructed by associating a vertex υ_i with each variable x_i that occurs in an inequality.
- The edge set, E, is composed of directed edges e_i. An edge is drawn for each individual inequality of the form of (5.1), starting from x_i and ending at x_j. The edge weight, w, is equal to the constraint value such that $w(\upsilon_i, \upsilon_j) = d_{ij}$.
- There is a source vertex, υ_0, located at $x = 0$. An edge is drawn between the source vertex, υ_0, and all vertices that do not have any other vertices constraining them from their left side.

- There is an end vertex, υ_n, located after the last edge in the layout. An edge is drawn between all vertices that do not have any other vertices constraining them from their right side and the end source vertex, υ_n.
- All layout elements are assumed to have positive x-coordinates.

Following the above rules, the constraint-graph for the layout of Fig. 5.2a is shown in Fig. 5.2b. A constraint-graph composed only of minimum-distance constraints has no *cycles* [11]. It is usually called a directed acyclic graph (DAG).

Starting from the source vertex υ_0, there might exist several paths to reach a specific vertex υ_i. By taking the *longest* path from υ_0 to υ_i, summing all path weights and assigning the result to x_i, one makes sure that all inequalities in which x_i participates are satisfied. Therefore, the length of the longest path from υ_0 to υ_i gives the minimal possible x-coordinate value, x_i, of the vertex υ_i. This is the main idea of the longest-path algorithm used in solving the constraint-graph [14], i.e., finding the optimum x position of all elements for minimal area.

The main advantage of using the constraint-graph and the associated longest-path algorithm is the computational simplicity and efficiency. However, it suffers from two main drawbacks: First, all elements are pushed as close as possible to the left boundary. Figure 5.3 shows the final compaction result after applying the longest-path algorithm on the layout of Fig. 5.2a. It is clear that the gate contact together with the enclosing metal have moved to the leftmost edge of the layout, causing a well-known problem of the longest-path algorithm referred to as *long wires* . Besides damaging layout shape and increasing the associated parasitics, long wires also affect subsequent compaction in the other direction. This problem was addressed in [15, 16].

The second main drawback is a fundamental one related to the graph itself. By construction, the graph can support constraints with only two variables. Two-variable constraints, as given by (5.1), represent the majority of layout constraints. However, other types of constraints with more than two variables are essential while describing matching and symmetry [12]. They have the general form of

$$x_b - x_a = x_d - x_c \qquad \text{where } a, b, c, d \in (1 \ldots k). \tag{5.2}$$

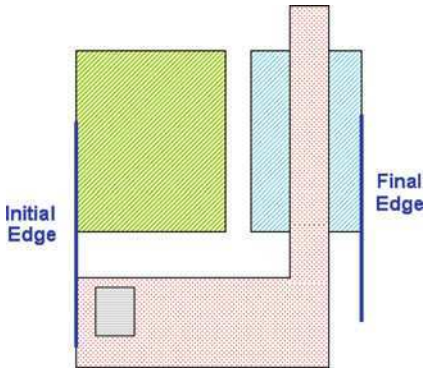


Fig. 5.3 Long wire problem

In addition, while dealing with hierarchical layouts, the number of variables increases with the number of hierarchy depth involved [17]. To support more general constraints with any number of variables, a more general approach has been introduced. This is discussed in the following section.

5.4.2 Linear Programming: The Simplex Method

One-dimensional layout compaction can be formulated in two separate linear programming (LP) optimization problems [17], one in the x and the other in the y-dimension. As in the previous section, the decision variables are the x-coordinates of all layout elements: $x_1, x_2, \ldots, x_k$. The goal function to be minimized (or maximized) is a *linear* objective function, f, in the above decision variables. This function is subjected to a set of *linear* constraints in the general form of

$$f_i(x_0, x_1, \ldots, x_k) \begin{pmatrix} \leq \\ = \\ \geq \end{pmatrix} d_{ij}. \tag{5.3}$$

The majority of such constraints are minimum-distance separation constraints in the form of (5.1). If constraints include more than two variables, they will be referred to as *multivariable* or *nondistance* constraints. It is preferable to formulate all such constraints as less-than by using some equation manipulations [18]. Then, for m constraints, the linear programming problem can be defined as follows:

$$\begin{aligned} \text{Minimize} \quad & f(x_1, x_2, \ldots, x_k) \\ \text{Subject to:} \quad & f_1(x_1, x_2, \ldots, x_k) \leq d_{f1} \\ & f_2(x_1, x_2, \ldots, x_k) \leq d_{f2} \\ & \vdots \\ & f_m(x_1, x_2, \ldots, x_k) \leq d_{fm} \\ & x_1, x_2, \ldots, x_k \geq 0 \end{aligned} \tag{5.4}$$

This kind of problems is often solved using the *simplex method* [18], which is briefly explained in the rest of this section.

Starting from (5.4), all constraints are transformed to equal constraints by adding *slack* variables. If there are k decision variables and m constraints, by adding a slack variable in each constraint, the constraint equations become:

$$x_{k+i} = d_{fi} - f_i(x_1, x_2, \ldots, x_k) \qquad \text{where } 1 < i < m \tag{5.5}$$

where m slack variables $(x_{k+1}, x_{k+2}, \ldots, x_{k+m})$ are introduced by extending the location variable set. This is a common practice, since during the simplex method,

all variables are subject to the same kind of operations. The problem can then be put in the matrix form to become:

$$\begin{aligned} \text{Minimize} \quad & f = \mathbf{c}_x^T \mathbf{x} \\ \text{Subject to:} \quad & \mathbf{A}\mathbf{x} = \mathbf{d} \\ & \mathbf{x} \geq 0 \end{aligned} \tag{5.6}$$

The vector $\mathbf{x}$ is an $(n \times 1)$ vector containing all variables (both location and slack variables), where $n = k + m$ is the total number of variables. The vector $\mathbf{c}_x$ is an $(n \times 1)$ vector containing all variable coefficients in the objective function, f. The matrix $\mathbf{A}$ is an $(m \times n)$ matrix containing all variable coefficients in each constraint equation. The vector $\mathbf{d}$ is an $(m \times 1)$ vector containing all constant terms in the constraint equations.

Definition 5.1 (Basic feasible solution). A basic feasible solution (BFS) is a solution with m constraints and n variables that

- Satisfies *all* m constraints,
- Includes m *basic* variables with values greater than or equal to zero, and
- Includes k $(= n - m)$ *nonbasic* variables with zero value.

The simplex algorithm starts from a BFS and iterates to other *better* feasible solutions, in the sense that they have smaller objective function values, until an optimal solution is reached. To show how to proceed from one iteration to the following one, some mathematical manipulations are performed on the simplex problem definition given by (5.6) to separate basic variables from nonbasic ones:

- The m basic variables are stored in the vector $\mathbf{x}_B$ while the k nonbasic variables are stored in the vector $\mathbf{x}_N$.
- Similarly, the vector $\mathbf{c}_x$ is split into the vectors $\mathbf{c}_B$ and $\mathbf{c}_N$, where $\mathbf{c}_B$ contains the coefficients of the basic variables in $\mathbf{x}_B$, while $\mathbf{c}_N$ contains the coefficients of the nonbasic variables in $\mathbf{x}_N$.
- In the same way, the $\mathbf{A}$ matrix is split into two matrices: matrix $\mathbf{B}$, containing all columns in $\mathbf{A}$ associated with the basic variables, and matrix $\mathbf{N}$ containing all columns in $\mathbf{A}$ associated with the nonbasic variables.

Using this separation, the simplex problem defined by (5.6) becomes:

$$\begin{aligned} \text{Minimize} \quad & f = \mathbf{c}_B^T \mathbf{x}_B + \mathbf{c}_N^T \mathbf{x}_N \\ \text{Subject to:} \quad & \mathbf{B}\mathbf{x}_B + \mathbf{N}\mathbf{x}_N = \mathbf{d} \\ & \mathbf{x}_B, \mathbf{x}_N \geq 0 \end{aligned} \tag{5.7}$$

Both the basic variables vector, $\mathbf{x}_B$, and the objective function, f, are expressed in terms of the nonbasic variables vector, $\mathbf{x}_N$, giving:

$$\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{d} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N \tag{5.8}$$

$$f = \mathbf{z}^T \mathbf{x}_N + \mathbf{c}_B^T \mathbf{B}^{-1}\mathbf{d}, \tag{5.9}$$

where $\mathbf{z}$ is defined as the cost rate vector of nonbasic variables. It represents the rate of change or relative cost of f with respect to each nonbasic variable in $\mathbf{x}_N$, given by:

$$\mathbf{z} = \mathbf{c}_N - (\mathbf{B}^{-1}\mathbf{N})^T \mathbf{c}_B \tag{5.10}$$

in a given BFS, since the value of all nonbasic variables are equal to zero, the basic variables can be calculated using (5.8) to give:

$$\mathbf{x}_B^* = \mathbf{B}^{-1}\mathbf{d} \tag{5.11}$$

Therefore, as indicated by Definition 5.1, in a given BFS:

- All m constraints are satisfied,
- The basic variables are given by (5.11), and
- All nonbasic variables are equal to zero.

The simplex method starts with an initial BFS, all following iterations and eventually the final optimum solution must also lead to a BFS. Looking back at (5.9), the only way to decrease f is to select a nonbasic variable from $\mathbf{x}_N$ such that it has a negative coefficient in $\mathbf{z}$, then to increase it from zero to any positive value. Since this variable is now nonzero, it becomes a basic one. However, according to the BFS definition, the number of both basic and nonbasic variables are fixed. Therefore, one of the basic variables must drop to zero and replaces this newly changed variable in the nonbasic variable set. In summary, to move to the next BFS iteration in the simplex method, exactly one nonbasic variable becomes a basic one, called the *entering* variable. In the same time, exactly one basic variable drops to zero and becomes a nonbasic one, called the *leaving* variable. The steps of the simplex method can be summarized as follows, refer to Fig. 5.4:

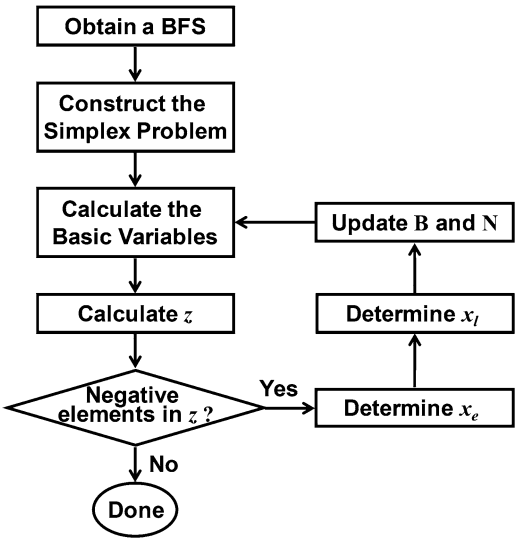


Fig. 5.4 Flow diagram of the simplex method

1. Obtain an initial BFS, refer to Sect. 5.5.2.3.
2. Construct the simplex problem in the form of (5.7).
3. Calculate the basic variable values using (5.11).
4. Calculate the cost rate vector, $\mathbf{z}$, for the nonbasic variables in the objective function, f, using (5.10).
5. Using $\mathbf{z}$, determine the *entering variable*, x_e, as follows: From the set of nonbasic variables, if all the corresponding coefficients in $\mathbf{z}$ are positive then f cannot be minimized anymore. Otherwise, the nonbasic variable with the most negative relative cost is selected to be the entering variable.
6. Determine the leaving variable, x_l, from the set of basic variables as follows: According to (5.8) and (5.11), the basic variables, $\mathbf{x}_B$, can be expressed in terms of their values at the previous BFS, $\mathbf{x}_B^*$, as follows:

$$\mathbf{x}_B = \mathbf{x}_B^* - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N \tag{5.12}$$

The vector $\mathbf{x}_N$ will have all values set to zero except at the entering variable position. The entering variable value will increase from zero to a certain positive value, t, such that

$$\mathbf{x}_N = [0, \ldots, 0, t, 0, \ldots 0]^T = t\mathbf{a}_e, \tag{5.13}$$

where $\mathbf{a}_e$ is a unit vector with all elements are zero except at the location of x_e. The t value should be as large as possible to minimize f, but at the same time it should maintain the nonnegativity condition of the basic variables as given by (5.12). Therefore,

$$\begin{aligned} \mathbf{x}_B &= \mathbf{x}_B^* - \mathbf{B}^{-1}\mathbf{N}\mathbf{a}_e t \\ &= \mathbf{x}_B^* - \Delta\mathbf{x}_B t \geq 0, \end{aligned} \tag{5.14}$$

where the step vector $\Delta\mathbf{x}_B$ expresses the rate of change of each basic variable when moving from a given BFS toward the next one in the solution space, and is given by:

$$\Delta\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{N}\mathbf{a}_e. \tag{5.15}$$

The leaving variable is the first variable that will become zero while increasing t. Therefore, from (5.14), the following condition should be satisfied

$$x_i^* - \Delta x_i t \geq 0 \qquad \forall x_i \in \mathbf{x}_B, \tag{5.16}$$

from which, the positive variable t should be selected as large as possible satisfying the condition:

$$t \leq \frac{x_i^*}{\Delta x_i} \qquad \forall x_i \in \mathbf{x}_B \tag{5.17}$$

This ratio is calculated for all basic variables, with t acquiring the minimum value of the whole set. The corresponding basic variable that has this minimum ratio will reach zero after increasing the entering variable to t. It is then assigned to be the leaving variable. In other words, the leaving variable is the one satisfying the condition:

$$\frac{x_l^*}{\Delta x_l} = \min\left\{\frac{x_i^*}{\Delta x_i} \qquad \Delta x_i > 0, 1 \leq i \leq m\right\} \tag{5.18}$$

7. Update the matrices **B** and **N** to reflect the new basic variable, x_e, and the new nonbasic variable, x_l.
8. Go to step 3. Repeat until an optimal solution is reached in step 5.

It is noted that the simplex method is efficient as long as the number of variables and constraints remain limited. The main resources and time intensive operation is that of finding the inverse of the **B** matrix as required in (5.8), (5.9), and (5.10). The inverse matrix, $\mathbf{B}^{-1}(i)$, at iteration i can be used to calculate the inverse of **B** at iteration $i + 1$. Some techniques such as LU factorization [18] can render the inverse matrix calculations more efficient in sparse matrices, where most of the matrix elements are zeros. This is somewhat true when the number of minimum-distance constraints of the form of (5.1) is large compared to multivariable constraints of the general form of (5.3). However, given the complexity of nowadays industrial layouts and the associated design rules, applying the pure simplex method turns out to be very time inefficient compared to graph-based techniques.

5.4.3 Graph-Based Simplex Methods

As a compromise of the aforementioned techniques, several methodologies were introduced to solve the LP problem of layout compaction using graph techniques, which are normally much faster [16, 19–21]. Such methodologies utilize the fact that most constraints are in the form of minimum-distance separation constraints, refer to (5.1). The number of other multivariable constraints, if they exist, is much smaller than the number of minimum-distance constraints. Also, these methods limit the shape of the objective function to be able to solve the problem using graph operations.

Marple et al. [16] introduced a graph-based simplex method, which only supported minimum-distance constraints, in addition to a special optimization function that minimizes long wiring lengths. Based on that, Onozawa [19] proposed an efficient graph-based algorithm that supports not only distance constraints but also multivariable constraints with a limited number of three variables.

A more general graph-based method that supports multivariable constraints was introduced by Wang and Lai [20]. It uses graph operations to speed up the calculations of the inverse of the basis matrix, **B**. The same constraint-graph is used, while applying graph operations to calculate the elements of a reduced core matrix. Matrix

operations are performed on the core matrix instead of the large basis matrix. As the size of the core matrix is proportional to the number of multivariable constraints, this method outperforms the simplex method as long as the number of multivariable constraints is lower than the number of minimum-distance constraints. However, a problem exists in getting an initial feasible solution. The initial solution is calculated using an algorithm described in [21] for only a set of multivariable constraints representing path delays in the layout. It lacks a procedure for getting a general initial solution for generic multivariable constraints. At the same time, the optimization function is restricted to contain only one variable. More complex objective functions are handled by adding more multivariable constraints.

As a conclusion, graph-based methods still remain limited either in the form of objective function or in the form of the constraints that are supported by graph operations. The multivariable constraint-graph based simplex method presented in the next section alleviates both of these limitations.

5.5 Multivariable Constraint-Graph Based Simplex Method

This section includes the core of this chapter, namely a graph-based simplex method that supports all forms of linear objective functions and linear constraints. This graph-based method combines both the efficiency of graph-based techniques and the generality of the simplex method. First, the graph definition is presented followed by details of the algorithm.

5.5.1 Basic Coefficient Constraint-Graph

A new graph representation, referred to as *coefficient Constraint-graph*, is introduced to model layout *locations* and multivariable *constraints*. The main idea behind this graph is to represent both constraints and variables as a signal flow graph that maintains the relation between them, while being general enough to handle any type of constraint.

Definition 5.2. Coefficient constraint-graph: the coefficient constraint-graph is constructed based on the following rules:

- Each variable, x_i, including slack variables, is represented as a graph node.
- All constraints, c_j, are also represented by separate nodes.
- For a constraint c_j composed of p variables

$$c_j : \quad f(x_1, x_2, \ldots, x_i, \ldots, x_p) = 0 \tag{5.19}$$

  a weighted directed arc from a variable node, x_i, to the constraint node, c_j, represents the coefficient of the variable in such constraint.
- A directed arc can exist only from a variable node to a constraint node.

- Since constraints usually have a constant term, an additional variable node, x_{bias}, is added to account for this term. x_{bias} is referred to as the *bias node*. By definition, it has the value of unity. A weighted arc connecting x_{bias} to a constraint represents the bias or the constant term of the corresponding equation. The general form of a constraint becomes:

$$c_j : \quad f(x_1, x_2, \ldots, x_i, \ldots, x_p, x_{\text{bias}}) = 0 \tag{5.20}$$

- Another type of nodes is the *supernode* . It consists of a constraint node associated with a variable node, (c_j, x_k). The connection coefficient between the variable node and the constraint node inside a supernode should always be equal to -1. The (c_j, x_k) supernode represents an equation in the form

$$c_j : \quad x_k = f(x_1, x_2, \ldots, x_i, \ldots, x_p, x_{\text{bias}}) \quad k \in 1 \ldots p \tag{5.21}$$

- From (5.21), it is clear that the weight of the coefficient arc between a variable x_i and a supernode (c_j, x_k) represents $\partial x_k / \partial x_i$.
- An arc starting from a supernode $[c_j, x_i]$ to a second supernode $[c_k, x_l]$ represents the coefficient of the variable of the first supernode, x_i, in the constraint of the second supernode, c_k. This means that the supernode acts as a variable node of its variable to successive nodes in the graph.

For example, the coefficient graph of the constraint

$$c_0 : \quad x_1 - x_0 - s_0 = 0$$

is shown in Fig. 5.5a. Note that all variables are moved to the left-hand side of the constraint. Another example with a bias node can be shown using the constraint

$$c_1 : \quad x_2 - x_1 - s_1 = 5$$

To represent the constant term in the above equation, the constraint can be expressed as

$$c_1 : \quad 5x_{\text{bias}} - x_2 + x_1 + s_1 = 0$$

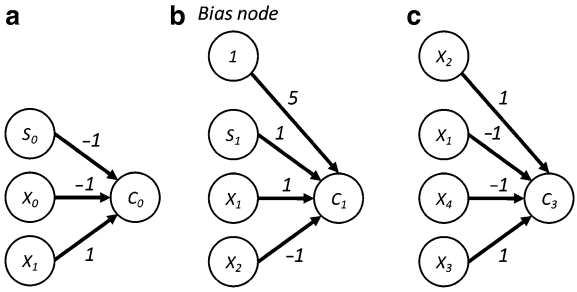


Fig. 5.5 Coefficient constraint-graph

The corresponding coefficient graph is depicted in Fig. 5.5b. Note that when the constant term is zero, as in c_0, no connection arc exists between the bias node and the constraint c_0. Consider now the symmetry constraint

$$c_3: \quad x_2 - x_1 = x_4 - x_3$$

It can be rewritten as

$$c_3: \quad x_2 - x_1 - x_4 + x_3 = 0$$

such that it can be represented by the coefficient constraint-graph shown in Fig. 5.5c. As an example of a supernode, the constraint c_0, can be represented as

$$c_0: \quad x_1 = 0 + x_0 + s_0$$

This representation is actually the *signal flow graph* of this constraint. In this case, the variable x_1 is associated with the constraint node to form one supernode as shown in Fig. 5.6. The weight of the coefficient connecting the node s_0 to the supernode (c_0, x_1) is actually equal to the derivative $\partial x_1/\partial s_0 = 1$. The same for x_0. In conclusion, the coefficient constraint-graph can represent any type of linear constraint with unlimited number of variables.

Going back to the simplex problem definition in terms of basic and nonbasic variables as given by (5.7), some modifications are needed to include the bias variable, x_{bias}, to account for the constant term in each constraint. Since the value of this variable is always set to unity, it can be added to the constraint equation as follows:

$$\mathbf{B}\mathbf{x}_B + \mathbf{N}\mathbf{x}_N = \mathbf{d}x_{\text{bias}} \tag{5.22}$$

Then, (5.9) and (5.14) for the objective function and basic variable vector become, respectively:

$$f = \mathbf{z}^T\mathbf{x}_N + \mathbf{c}_B^T\mathbf{B}^{-1}\mathbf{d}x_{\text{bias}} \tag{5.23}$$

$$\begin{aligned}\mathbf{x}_B &= \mathbf{x}_B^* x_{\text{bias}} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_N \\ &= \mathbf{x}_B^* x_{\text{bias}} - \Delta\mathbf{x}_B t.\end{aligned} \tag{5.24}$$

The simplex problem definition can now be represented using a *basic* coefficient constraint-graph.

Definition 5.3 (Basic coefficient constraint-graph). It is a coefficient constraint-graph as described by Definition 5.2, where each constraint node is associated with

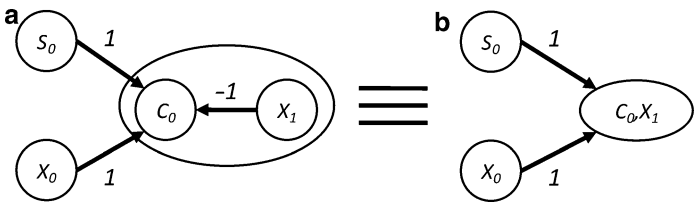


Fig. 5.6 Supernode $[c_0, x_1]$

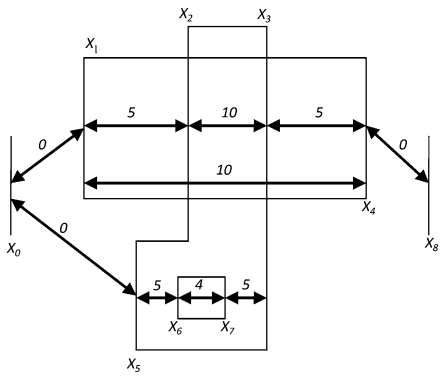


Fig. 5.7 Sample layout with sample design-rule constraints

a *basic* variable forming a supernode. An additional supernode is added, which contains both the objective function, f, and a constraint representing the corresponding equation.

For example, consider the layout shown in Fig. 5.7, the following inequalities represent the shown minimum-distance constraints:

$$\begin{aligned} c_0 &: x_1 - x_0 \geq 0 & c_5 &: x_5 - x_0 \geq 0 \\ c_1 &: x_2 - x_1 \geq 5 & c_6 &: x_6 - x_5 \geq 5 \\ c_2 &: x_3 - x_2 \geq 10 & c_7 &: x_7 - x_6 \geq 4 \\ c_3 &: x_4 - x_3 \geq 5 & c_8 &: x_3 - x_7 \geq 5 \\ c_4 &: x_4 - x_1 \geq 10 & c_9 &: x_8 - x_4 \geq 0 \end{aligned} \tag{5.25}$$

where all c_j's represent distance constraints containing the x_i location variables. As in the simplex method, all constraints should be equal constraints by introducing slack variables, refer to (5.5). Also, all constant terms are multiplied by the bias variable, x_{bias}. Assume that the initial BFS contains the following basic variable set: $x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, s_4, s_8$. Note that the number of basic variables is equal to the number of constraints given in (5.25). Also, assume that the objective function is given by the following equation:

$$f = x_8 - x_5 - x_0 \tag{5.26}$$

then, the simplex problem can be formulated as follows:

$$\begin{aligned} \text{Minimize}: \quad & f = x_8 - x_5 - x_0 \\ \text{Subject to}: \quad & c_0 : x_1 = 0 + x_0 + s_0 & c_5 &: x_5 = 0 + x_0 + s_5 \\ & c_1 : x_2 = 5x_b + x_1 + s_1 & c_6 &: x_6 = 5x_b + x_5 + s_6 \\ & c_2 : x_3 = 10x_b + x_2 + s_2 & c_7 &: x_7 = 4x_b + x_6 + s_7 \\ & c_3 : x_4 = 5x_b + x_3 + s_3 & c_8 &: s_8 = -5x_b + x_3 - x_7 \\ & c_4 : s_4 = -10x_b + x_4 - x_1 & c_9 &: x_8 = 0 + x_4 + s_9 \end{aligned} \tag{5.27}$$

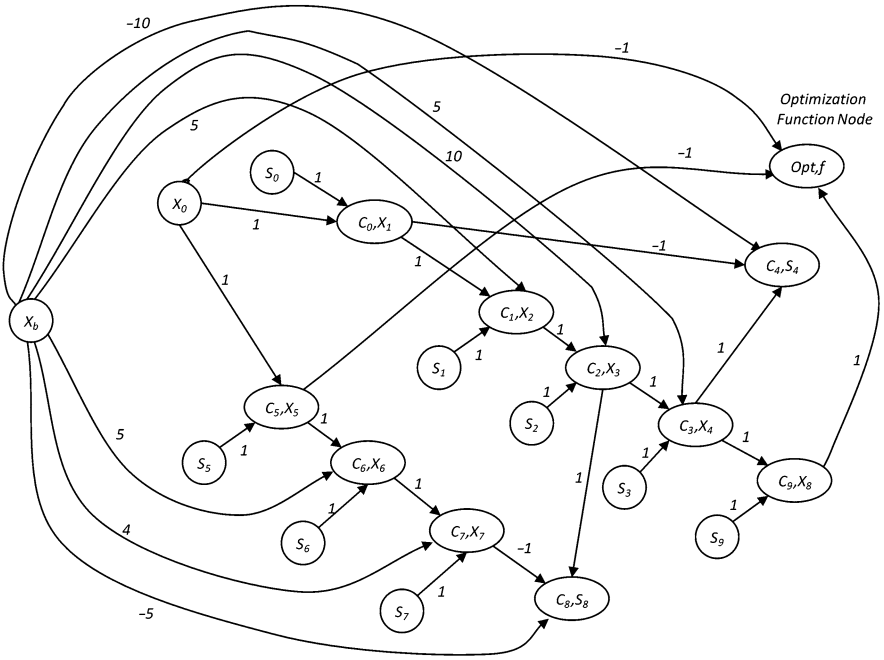


Fig. 5.8 Basic coefficient constraint-graph

The resulting basic coefficient constraint-graph of the above problem is shown in Fig. 5.8. The number of supernodes is equal to the number of constraints in addition to an optimization function supernode. The next section shows how to use such graph to solve the optimization problem.

5.5.2 *Multivariable Graph-Based Simplex Algorithm*

The main idea of the Multivariable Constraint-Graph algorithm is to employ the graph, described in Sect. 5.5.1, to replace complex matrix operations of the simplex method, described in Sect. 5.4.2. The basic coefficient constraint-graph is the signal flow graph representing algebraic equations of constraints and a given objective function. The rules of the basic coefficient constraint-graph are those of a signal flow graph [22]:

Gain Equation

For any two nodes x_i and x_j, the derivative $\partial x_j / \partial x_i$ can be found by Mason's equation [22]. In case there are no loops, it reduces to

$$\partial x_j / \partial x_i = \sum_i G_i, \tag{5.28}$$

where G_is are the gains of all paths from x_i to x_j. Equation (5.28) also defines the *connection value* between any two nodes x_i and x_j. As will be shown, the connection value between graph nodes can be used to replace the corresponding matrix operations.

Graph Mathematics

In what follows, a set of rules are introduced to obtain the required simplex method vectors in terms of graph *connection values*. All rules assume that there are no loops in the basic coefficient graph:

Rule 1 From (5.24), it is clear that in a given BFS, the basic variables matrix, $\mathbf{x}_B^*$, can be deduced from the general basic variable equation as follows:

$$\mathbf{x}_B^* = \partial \mathbf{x}_B / \partial x_{\text{bias}} \tag{5.29}$$

Hence, the individual basic variables in a given BFS can be calculated using

$$x_{\text{basic}}^*(i) = \partial x_{\text{basic}}(i) / \partial x_{\text{bias}}, \tag{5.30}$$

Therefore, in a given BFS, the basic variable value, $x_{\text{basic}}^*(i)$, is the *connection value* between the x_{bias} variable node and the $x_{\text{basic}}(i)$ basic variable supernode in the basic coefficient graph.

Rule 2 From (5.23), the relative cost rate vector $\mathbf{z}$ can be expressed as

$$\mathbf{z} = \partial f / \partial \mathbf{x}_N, \tag{5.31}$$

Therefore, the value of the cost rate vector element, $z(i)$, is the connection value between the $x_{\text{nonbasic}}(i)$ variable node and the f optimization function supernode in the basic coefficient graph.

Rule 3 The step vector $\Delta \mathbf{x}_B$ can be calculated from (5.24). Since t is the value of the entering variable x_e, then

$$\Delta \mathbf{x}_B = -\frac{\partial \mathbf{x}_B}{\partial t} = -\frac{\partial \mathbf{x}_B}{\partial x_e}, \tag{5.32}$$

Therefore, the value of the step vector element, $\Delta x_{\text{basic}}(i)$, is equal to -1 multiplied by the connection value between the entering variable node and the $x_{\text{basic}}(i)$ variable supernode in the basic coefficient graph.

The steps of the multivariable graph-based simplex algorithm are the same as those of the simplex method presented in Sect. 5.4.2, refer to Fig. 5.4. However,

each step is performed using simple graph operations instead of complex matrix ones. The proposed algorithm would go through the following steps:

1. Obtain an initial BFS, refer to Sect. 5.5.2.3.
2. Construct the corresponding basic coefficient constraint-graph.
3. Calculate the basic variables values, $x^*_{\text{basic}}(i)$. This is done using Rule 1, (5.30), by calculating the connection value from the bias node, x_{bias}, to all supernodes, which contain the basic variables.
4. Calculate the relative cost value, $z(i)$, of each nonbasic variable in the objective function, f. This is done using Rule 2, (5.31), by calculating the connection value from all nonbasic variable nodes to the objective function supernode, f.
5. Using $z(i)$, determine the *entering variable*, x_e, as follows: From the set of nonbasic variables, if all the corresponding values in $z(i)$ are positive then f cannot be minimized anymore. Otherwise, a nonbasic variable with the most negative relative cost is selected to be the entering variable.
6. Determine the leaving variable, x_l, from the set of basic variables using (5.18). To calculate the step vector elements $\Delta x_{\text{basic}}(i)$, (5.32) of Rule 3 is applied. The rate of change of each basic variable with respect to the entering variable, x_e, is calculated by determining the connection value from x_e to all basic variable supernodes, then multiplying it by -1.
7. Update the graph so that the entering variable, x_e, is associated with a constraint node forming a supernode, while the leaving variable, x_l, is deattached from its supernode to become a nonbasic variable in the new basic constraint-graph, refer to Sect. 5.5.2.1.
8. Go to step 3. Repeat until an optimal solution is reached in step 5.

5.5.2.1 Updating the Basic Coefficient Constraint-Graph

During a given iteration, after selecting both entering and leaving variables, the graph should be updated to prepare for the next iteration. The leaving variable will become a nonbasic variable, so it should be dissociated from any supernode. On the other hand, the entering variable should be associated with a constraint to form a supernode. For example, in the first iteration of the graph shown in Fig. 5.8, assume that s_5 is the entering variable while s_8 is the leaving one. The graph should be changed such that s_5 becomes associated with a constraint and s_8 should be detached from its current supernode.

To update the graph, a path from the entering variable to the leaving one should be established. In the layout example, this path is shown in Fig. 5.9a. After finding the connecting path, all variables inside a supernode belonging to this path are detached from their supernode. Then, each variable in the new path is associated with its successive constraint in the path to form a new supernode. The new supernodes are shown in Fig. 5.9b. During the construction of new supernodes, the basic variable coefficients should be adjusted so that the coefficient connecting the constraint and the basic variable inside the supernode should always be equal to -1.

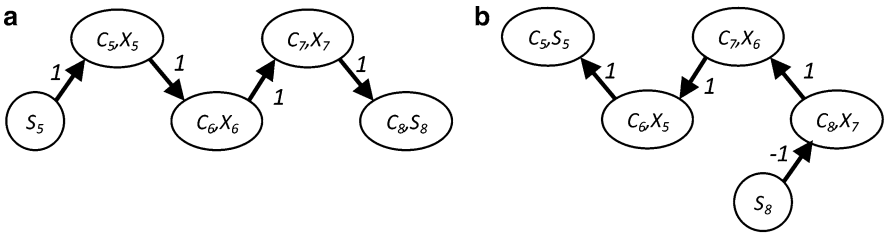


Fig. 5.9 The path between entering and leaving variables: (**a**) before the graph update, and (**b**) after the graph update

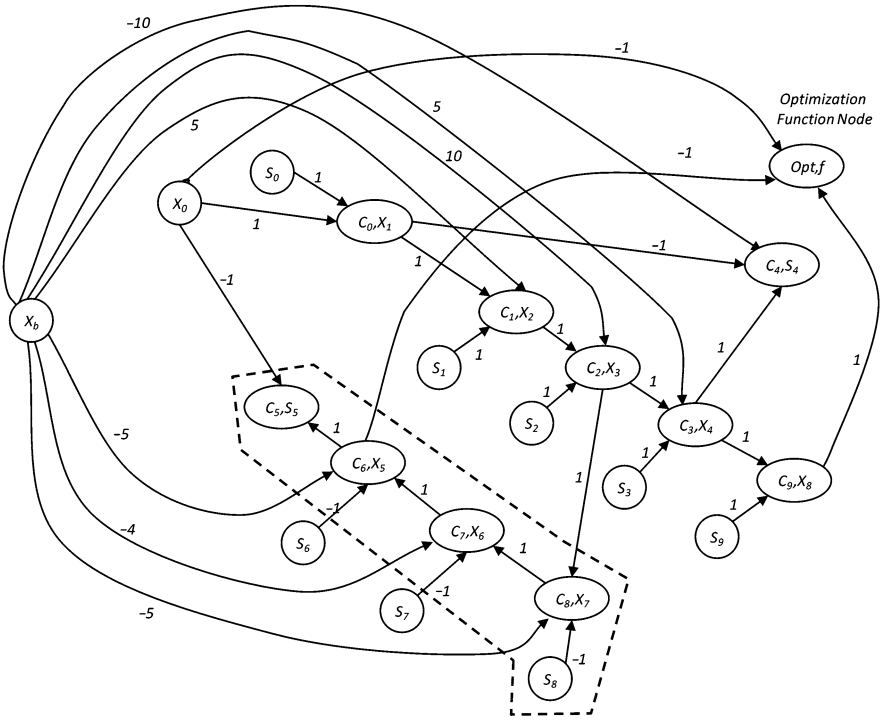


Fig. 5.10 Basic coefficient constraint-graph after the first iteration

The new basic coefficient constraint-graph of the second iteration is shown in Fig. 5.10. The updated nodes are marked with a dashed polygon. Only the coefficients connected to updated nodes are changed in the graph. This means that any coefficient arc connecting two nodes outside the updated area will stay intact during such update.

5.5.2.2 Dealing with Loops

The main assumption that allowed the simple application of Mason's formula, (5.28) is that there are no loops in the graph. However, sometimes loops can not be avoided. In this section, a method is presented to eliminate loops by successive substitutions of constraint equations. For example, if there exists a simple loop of three constraints: c_x, c_y, and c_z, such that

$$\begin{aligned} c_x &: x_6 = 5 + 4x_8 + s_6 \\ c_y &: x_7 = 4 + x_6 + s_7 \\ c_z &: x_8 = 0 + x_7 + s_8 \end{aligned}$$

These equations are represented in the graph shown in Fig. 5.11a. It should be noted that nonbasic variable nodes can never be inside a loop since they always have arcs going outward to a constraint supernode. Also, each basic variable (supernode) in a loop should appear in at least two constraints, its constraint supernode and the successive constraint supernode in the loop. Substituting by equations c_z and c_y in equation c_x to remove x_7 and x_8 basic variables from such constraint will produce

$$c_x : x_6 = -7 + -\frac{4}{3}s_7 - \frac{4}{3}s_8 - \frac{1}{3}s_6$$

as shown in Fig. 5.11b. It shows that the new coefficient graph representation has no loops. By the employment of the above strategy, any loop can be eliminated by successive substitution steps. However, this operation may result in an increasing number of coefficient links in the graph.

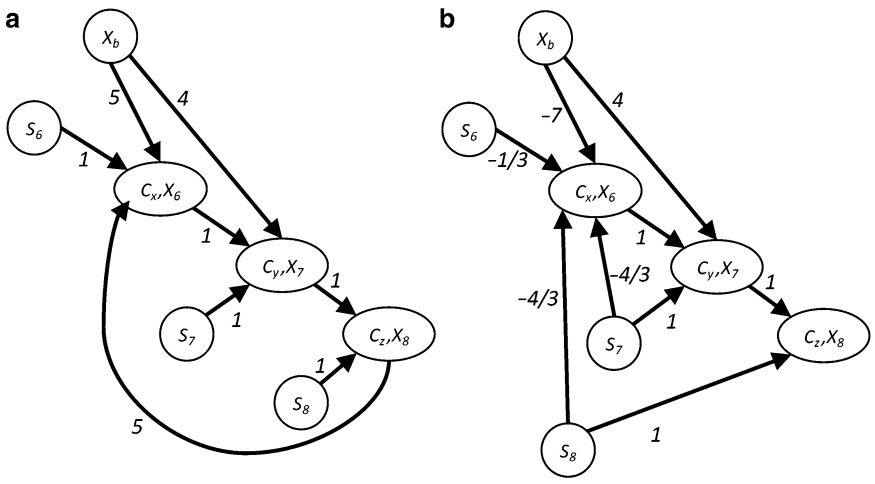


Fig. 5.11 Loop c_x, c_y and c_z (**a**) before and (**b**) after the elimination

Theorem 5.1. *Each loop in the basic coefficient constraint-graph contains at least one nondistance constraint.*

Proof. Slack variables of minimum-distance constraints, s_i, cannot appear in a loop since each slack variable appears only in one constraint. Assuming that there is a loop of k distance constraints containing the basic variables $x_1, x_2, \ldots, x_k$, the loop constraints can be expressed as:

$$\begin{aligned} x_2 &= x_1 - \rho_1 s_1 + \rho_1 d_1 \\ x_3 &= x_2 - \rho_2 s_2 + \rho_2 d_2 \\ &\vdots \\ x_k &= x_{k-1} - \rho_k s_{k-1} + \rho_{k-1} d_{k-1} \\ x_1 &= x_k - \rho_k s_k + \rho_k d_k \end{aligned} \tag{5.33}$$

where ρ_i can be either 1 or -1. By summing all the equations, this results in:

$$\rho_1 s_1 + \rho_2 s_2 + \cdots + \rho_{k-1} s_{k-1} + \rho_k s_k = \rho_1 d_1 + \rho_2 d_2 + \cdots + \rho_{k-1} d_{k-1} + \rho_k d_k. \tag{5.34}$$

Since all variables in the above equation are nonbasic ones, this row summing operation has canceled all basic variables. This means that in the corresponding basis matrix **B**, this summing can produce a row that contains only zeros in **B**, i.e., it is a singular matrix. This clearly contradicts the fact that **B** *is a nonsingular matrix* [18], i.e., it contains no linearly dependent rows. Therefore, the loop assumption of only minimum-distance constraints is not valid. So, no loops can contain only distance constraints. □

5.5.2.3 Initial Basic Feasible Solution

The initial point greatly affects the performance of the algorithm. If the initial solution is near the optimal point, only few iterations are needed to reach the optimal solution. One way to get an initial solution is to use the dual simplex method [18]. However, this may increase the number of iterations resulting in an effective reduction in performance.

A good initial solution can be obtained by applying the longest path algorithm, refer to Sect. 5.4.1. However, this algorithm only guarantees that all minimum-distance constraints with two variables are satisfied, but not the general multivariable ones. Hence, it does not produce a valid BFS. A solution to this problem was introduced in [21] based on the special form of signal delay constraints. However, in the general case, like in symmetry constraints, there is no guarantee to obtain a BFS by this method. Based on the same idea, a more general method to obtain an initial BFS is elaborated as follows:

1. The longest path algorithm is first applied. The obtained locations are used as the initial solution. All other variables that are not involved in any distance constraints are initialized to zero.
2. Each general multivariable constraint is checked. If the constraint is satisfied, then its constraint node is associated with its slack variable and no additional operation is needed.
3. If the general constraint is not satisfied or the constraint does not contain slack variables, e.g., symmetry constraint, an artificial variable is added. The value of this variable is chosen such that the new constraint is satisfied. These artificial variables should be zero in the final optimal solution. At the same time, a new term is added to the objective function, f, to penalize the added variables. Such new terms would be equal to the artificial variable itself multiplied by a huge positive constant, M.

   As an example, suppose that there exists the following symmetry constraint:

$$x_4 - x_3 = x_2 - x_1 \tag{5.35}$$

If the x-values obtained by the longest path algorithm are:

$$x_1 = 10, \quad x_2 = 30, \quad x_3 = 10, \quad x_4 = 50$$

applying these values, the symmetry constraint is not satisfied. Now an artificial variable a_1 is added so that the constraint becomes:

$$x_4 - x_3 = x_2 - x_1 + a_1 \tag{5.36}$$

the value of a_1 is chosen to be 20 so that the new constraint is satisfied. At the same time, the term Ma_1 is added to the objective function, with a huge value of M. This ensures that the variable a_1 is reduced to zero in the final optimal solution such that the original constraint (5.35) is satisfied.

Another important type of constraints is equal-constraints of the form

$$x_j - x_i = K. \tag{5.37}$$

Such kind of constraints are essential to guarantee device dimensions are accurately sized in the layout. Other layout shapes, e.g., contacts and vias need to resize for well-defined dimensions to avoid design-rule violations. These constraints are handled similar to the symmetry constraints by adding an artificial variable both in the constraint and in the objective function and requiring that it drops to zero in the final solution.

It is worth mentioning that the addition of new terms in the new graph-based method is straightforward, which is not the case either in the constraint-graph method [11] or in the graph-based simplex method [20].

5.5.3 Complexity Analysis

In this section, the complexity of the multivariable graph-based method is compared to that of the original simplex one. It should be noted that a single iteration of the graph-based algorithm results in the same output produced by a corresponding iteration of the matrix-based simplex method. Hence, if started from the same BFS, both would have the same number of iterations. The complexity analysis is carried out for only a single iteration.

The complexity of a single simplex iteration is dominated by the inversion of the coefficient matrix, **B**, which is

$$C_{\text{simplex}} = O(R^3), \tag{5.38}$$

where R is the number of rows (or columns) of the basis matrix [20]. Some sparse matrix techniques such as LU factorization can help to reduce this complexity.

For constraint-graph techniques, the complexity of the whole longest path algorithm is

$$C_{\text{longest path}} = O(N + E), \tag{5.39}$$

where N is the number of graph nodes, and E is the number of links [23]. Practically, the longest path algorithm is $O(N)$.

Comparing (5.38) and (5.39) explains the huge performance difference between graph-based longest-path and simplex methods.

Multivariable Graph-Based Algorithm Complexity:

The complexity depends heavily on the kind of constraints present in the layout and the corresponding equations. Both have a direct effect on the graph shape, and whether there are existing loops or not.

Basically, there are two main operations in the multivariable graph-based method:

1. The step of finding the *connection value* between a node and all other nodes in the graph, i.e., steps 3, 4, and 6 in Sect. 5.5.2. This step has a similar complexity of the longest path algorithm given by (5.39) [24].
2. Dealing with loops that might appear due to multivariable constraints, refer to Sect. 5.5.2.2.

To estimate the complexity of each operation, let us assume the following:

- The number of location variables is k.
- The number of distance constraints is u.
- The number of multivariable nondistance constraints is v.

- The maximum number of variables in a nondistance constraint is p. This value may vary from five for a simple symmetry constraint of the form of (5.2), to tens of variables in case of multiple-level hierarchical layouts.
- The total number of constraints is $m = u + v$.
- After adding slack variables, the total number of variables will be $N = k + m = k + u + v$, refer to (5.5). Hence, the graph size (or the total number of nodes in the graph) is N.

The number of supernodes (or the number of basic variables) and the number of separate nonbasic variable nodes will be $m = u + v$ and k, respectively. For each distance constraint of the form $x_i - x_j - s_k = d_k$, there will be at most three coefficients, one for each variable besides the bias node minus the basic variable, refer to Fig. 5.8. Similarly, the number of links of nondistance constraints will be $(p-1)\times v$. Therefore, the total number of links in the graph will be $E = v\times$ (p-1)$+$ $3 \times u + k_f$, where k_f is the number of links associated with the objective function supernode. So, from (5.39), the complexity of finding connection values between a given node and all other nodes of the graph-based method would be

$$\begin{aligned} C_{\text{graph-connection value}} &= O(N + E) \\ &= O((k + u + v) + (p - 1) \times v + 3 \times u + k_f) \\ &= O(k + k_f + u + p \times v) \end{aligned} \tag{5.40}$$

However, depending on the constraint complexity, each nondistance constraint may be part of many loops. To eliminate all of these loops, the substitution technique mentioned in Sect. 5.5.2.2 should be used. The maximum number of substitution steps for each variable included in any nondistance constraint is equal to the total number of supernodes in the graph, $O(u + v)$. Since the maximum number of variables in a single nondistance constraint is p, then the worst-case complexity to eliminate all loops associated with one nondistance constraint would be

$$C_{\text{graph-single non-distance constraint loop elimination}} = O(p \times (u + v)). \tag{5.41}$$

In the worst case, such process is needed for *each* nondistance constraint to eliminate associated loops. Therefore, the complexity of all loop elimination would be

$$C_{\text{graph-loop elimination}} = O(v \times p \times (u + v)) \tag{5.42}$$

Now consider the following special cases:

- There are only distance constraints, i.e., $v = p = 0$ and $m = u$: Equation (5.40) becomes

$$C_{\text{graph-v=0}} = O(k + k_f + u) \tag{5.43}$$

which is a linear complexity with time. In this case, there are no loops in the graph as proved in Sect. 5.5.2.2. *Therefore, the whole graph-based simplex single iteration has the same linear complexity as the longest-path algorithm.*

- The number of distance constraints is much larger than the number of nondistance constraints, i.e., $u >> v$:
  This is the most common case since most of the constraints are of the minimum distance design rule of the form of (5.1). Equation (5.40) becomes

$$C_{\text{graph-connection value-}u >> v} = O(k + k_f + u), \tag{5.44}$$

  which is still a linear complexity with time. In this case, loops may appear, therefore (5.42) reduces to

$$C_{\text{graph-loop elimination-}u >> v} = O(v \times p \times u), \tag{5.45}$$

  which is still a manageable complexity, specially if the number of variables in nondistance constraints, p, is limited.
- A hypothetical limiting case for the algorithm in which the number of nondistance constraints is much larger than that of the distance constraints, i.e., $u << v$. In addition, the maximum number of variables in a nondistance constraint is equal to the total number of variables, i.e., $p = N = k + u + v \approx k + v$. Therefore, (5.40) and (5.42) become, respectively

$$\begin{aligned} C_{\text{graph-connection value-}u << v} &= O(k + k_f + v \times (k + v)) \\ &= O(v^2) \end{aligned} \tag{5.46}$$

$$\begin{aligned} C_{\text{graph-loop elimination-}u << v} &= O(v \times (k + v) \times v) \\ &= O(v^3). \end{aligned} \tag{5.47}$$

  Comparing the last complexity with (5.38), *this is the same complexity of matrix inversion in the simplex method iteration.*

If the matrix is a sparse matrix, the matrix inversion can be more efficient by using LU factorization. Similarly, the graph-based simplex algorithm becomes more efficient because the number of links in the graph decreases in a matrix with many zeros. Practically, the number of nondistance constraints, such as symmetry constraints, is small ($u >> v$) and the number of variables appearing in this constraints is very small compared to the total number of variables ($p << N$). Consequently, the complexity of the graph-based algorithm is very well below the hypothetical limiting case complexity. In other words, the algorithm benefits from the sparsity of matrices to reduce the computation time. This is an inherent property of the algorithm without the need for any special sparse handling. The graph size increases to be maximum when the connections are only between nonbasic variable source nodes and basic variable supernodes. The number of coefficients of a supernode is equal to the number of nonzero elements in the matrix row of the corresponding constraint. Hence, the size complexity of the graph is *at most* the same size complexity of the simplex method.

To enhance the efficiency of the multivariable graph-based algorithm, only a subset of nodes are processed. Those nodes are the updated nodes affected by the entering variable, as shown in Fig. 5.10, in addition to any node that has a path involving the objective supernode, f, and passing through any of the updated nodes. All remaining nodes will maintain the same values as the previous iteration.

5.6 Layout Constraints Revisited

As shown in Sect. 5.5.3, the compaction problem complexity is closely related to the number and kind of constraints in the corresponding layout. Both are closely related to the total number of elements and the detailed manner in which the layout of each is designed. Layout retargeting must keep all such details while going from one process to another. In this section, different aspects related to layout constraints are discussed: First, a method to reduce their number is introduced. Then, some issues related to the one-dimensional nature of constraints are presented.

Constraint Number Reduction

Layout simplification can be attained by merging some structures before compaction, then re-expanding them in the target process after compaction during a postprocessing step. Among the layout layers that add a lot of edges, and hence many constraints are the contact and via layers. Structures in such layers are characterized by their small size and large number. An example is the contact array found at each transistor source and drain areas. Another example are the via/contact arrays designed to handle high current densities as shown in Fig. 5.12a. In the figure, arrows indicate design-rule constraints related to the shown array composed of sixteen contacts. After migration, the space occupied by such arrays is susceptible to

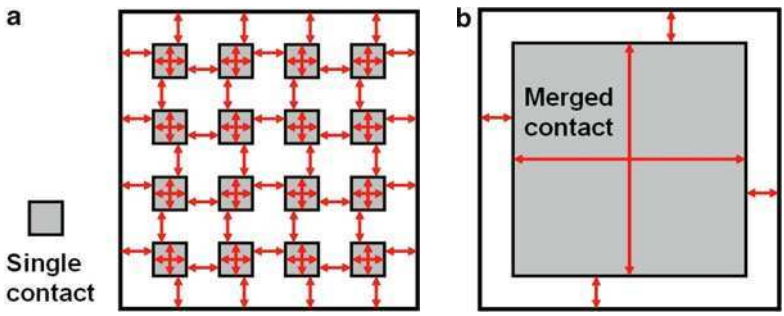


Fig. 5.12 Contact array (**a**) before merging and (**b**) after merging

change according to the target device sizes and current densities. This problem can be handled using the following three steps:

1. During layer mapping, see Sect. 5.3.1, each contact/via array is merged into one single large contact/via occupying the whole area of the array as shown in Fig. 5.12b. It is clear from the figure that the number of constraints is reduced considerably.
2. The internal design-rule constraints that determine the size of the single large contact in Fig. 5.12b is set to keep at least the same number of vias as in the source layout, unless otherwise specified.
3. After compaction, a postprocessing step reconstructs the minimum-sized contacts and vias from the corresponding large contact.

Complex Design-Rules

Traditionally, layout design-rules used to belong to either minimum width, minimum spacing, or minimum extension rules. Recently, design-rule complexity is increasing by moving from standard edge-based rules to shape or polygon-based ones. Moreover, some rules need to be specified by multivariable equations rather than a fixed value constraint. This makes well-known constraint generation methods, e.g., the scan line method [25], more challenging to apply. In fact, the one-dimensional nature of compaction puts some limitations on the kind of layout design-rules that can be handled. For example, dealing with *conditional* rules represents a real challenge [26]. Conditional design-rules are ordinary intralayer or interlayer design-rules in which the constraint parameter is conditional on some outside factor. For example, the metal separation rule has a larger value if one of the metals becomes a wide metal. Conjunctive or context-based design-rules represent a specific class of conditional design-rules in which the constraint parameter depends on the presence or absence of an otherwise unrelated layer. Attempts to deal with such rules have been made in [27, 28]. The use of the Calibre nmDRC™ [9] tool as mentioned in Sect. 5.3.3 easily captures such constraints, which are then transformed to edge-based format.

Edge Order

Reformulating design-rule constraints in edge-to-edge-based ones triggers additional challenges. For example, if two edges overlap in the source layout, i.e., they have a zero separation distance, while the target design-rules impose a minimum-distance constraint between them, the constraint generator will have much difficulty to assign an order for these edges. By default, minimum-distance constraints predictate a specific order that must be conserved. This might result in nonoptimal area in the target process. For example, the following constraint:

$$x_j - x_i \geq d_{ij} \tag{5.48}$$

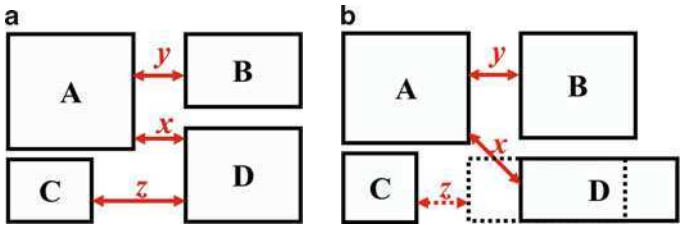


Fig. 5.13 Compaction edge order: (**a**) source layout and (**b**) after migration

means that, after compaction, the value of x_j will always be greater than that of x_i. In other words, the order of these edges cannot be reversed. In some cases, device sizes and aspect ratio can considerably change from one process to the other. An example is shown in Fig. 5.13a, which shows the placement of four devices: A, B, C, and D, with three horizontal distance constraints, namely: x, y, and z. After device size recalculations in the new process, device shapes are shown in Fig. 5.13b. The obtained device sizes would allow device D to move to the left up to the minimum of the z-constraint as shown by the dotted rectangle. However, due to the existence of constraint x between devices A and D, which now has no meaning, device D can not be moved to the left and the x-constraint is maintained. In fact, the compactor has no information on device relationships in the vertical direction during horizontal compaction and vice versa. Possible solution to such problem is to iterate with subsequent horizontal–vertical compaction trials till no more area reduction is achieved. Each time, new constraints have to be generated. Two-dimensional compaction techniques can also solve such problems [11]. However, such techniques are proved to be nondeterministic polynomial (NP)-complete, which means that the computation time increases exponentially with the problem size. Several heuristic techniques are used to reach a solution in an acceptable time [29]. However, for practical cases, this is still not feasible and most compaction tools opt for one-dimensional techniques.

Nanometer Process Effects

With the advent of new process technologies, more constraints need to be added than just design-rule ones to keep the same layout electrical performance. Some second-order layout effects become so important to be neglected. Most of these effects are still not accounted for in most analog design automation tools. For example, recent publications on shallow trench isolation (STI) stress and well proximity (WP) effects have demonstrated the profound impact of layout variations on transistor performance [4]. The fact that the layout migration methodology does keep the layout floorplanning, placement, and routing helps to mitigate such effects, if and only if they were already accounted for in the source layout. In some cases, well dimensions are increased to control the WP effect in the source layout. However, during migration, the target of the compaction module is to minimize the dimensions of all

polygons to the specified constraints. This calls for special handling of well structures. Wells not adjusted to minimum sizes can be detected in the source layout so that further manipulations become possible. Compaction objective functions that privilege closeness to the source layout [2] can also help reduce such effects.

5.7 Practical Retargeting

Figure 5.1 shows the *basic* modules of a compaction-based layout retargeting tool. This section discusses other complementary modules that are needed to be able to handle industrial-level layout complexities. However, implementation details of such modules are out of the scope of this chapter. The main target of this section is to show that the multivariable graph-based simplex method presented in Sect. 5.5 allows the seamless integration of such modules in the automatic retargeting tool.

5.7.1 Symmetry Enforcement

Since the absolute value of integrated circuit components has large tolerances due to process and temperature variations, successful analog design is usually based on relative component accuracy rather than absolute one. Relative accuracy is strongly related to layout matching and symmetry techniques. During retargeting, such layout strategies must be *detected* in the source layout and *replicated* in the target one.

Symmetry constraints can be identified manually by the user for small layouts. For large hierarchical layouts, multilevel symmetry detection has been handled in [3] by utilizing the inherent circuit structure and hierarchy information from an extracted netlist.

The enforcement of such constraints has been problematic to compaction algorithms. As stated in Sect. 5.4.1, efficient graph techniques do not support symmetry constraints of the form of (5.2). This is one of the main reasons that recent implementations moved to LP techniques in spite of their excessively high computational cost, refer to Sect. 5.4.2. As a compromise, in [12] Okuda et al. relied on graph-based techniques to solve the main problem, while for symmetry constraints, another equivalent reduced LP problem that contains lower number of constraints is constructed. The reduced problem is then solved using the revised simplex method [30]. Revised simplex is a way of ordering the computations of the simplex method to avoid unnecessary calculations. Once symmetry constraints are satisfied, they are converted to more simple distance constraints and reflected back on the main large graph. However, still more general constraints with many variables cannot be handled. Moreover, The optimization function is restricted to a very simple form.

The presented multivariable graph-based algorithm would overcome all such problems since it supports any kind of layout constraints. In addition, it allows complex objective functions to be used in the same time.

5.7.2 Device Aspect Ratio

During analog circuit migration, device dimensions are subject to large variations with respect to their original sizes. Even if the same electrical performance is conserved, device characteristics change across different process technologies can lead to large device area variations. The described methodology aims to keep the same device placement and routing, in addition to all device related physical parameters, such as the number of fingers for each transistor, the number of unit resistors, and capacitors, etc. The initial relative device placement and orientation is normally optimized for the source layout. After the calculation of new device sizes, this might not be the optimum placement, which might lead to some waste of area after migration. A device outline example is shown in Fig. 5.14a. After migration, both device area and aspect ratio change to yield the result shown in Fig. 5.14b for the same relative placement. It is clear that a considerable area loss exists at the top-right and bottom-left corners. Using the same devices, area optimization is possible by just moving device *A* to the top-left corner as shown in Fig. 5.14c. This result is possible only using a two-dimensional compactor. However, as mentioned in Sect. 5.6, these techniques prove to be difficult to apply.

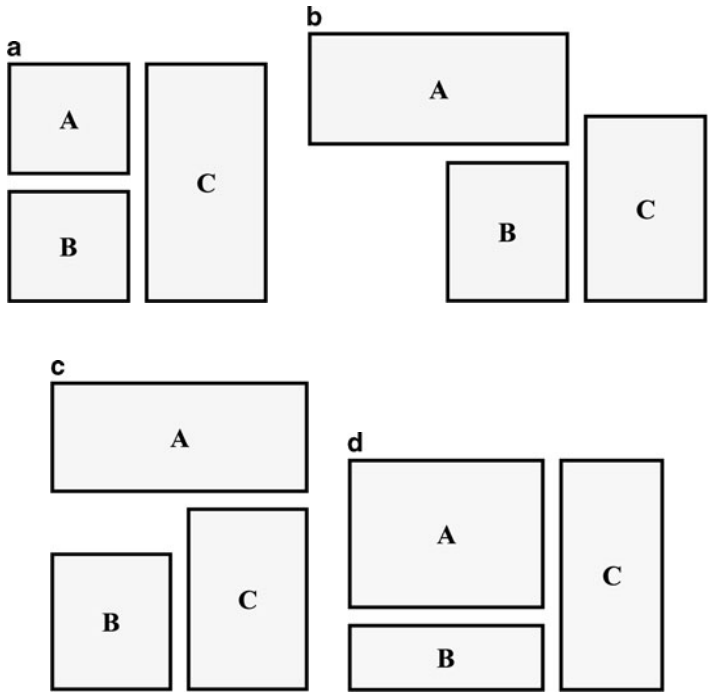


Fig. 5.14 Layout shape optimization: (**a**) source layout, (**b**) migrated layout, (**c**) migration using 2-D compaction, and (**d**) using device aspect ratio control

In [31], device aspect ratio is changed while keeping the same device relative placement. This can be done by investigating different realizations for each device, e.g., varying the number of transistor fingers. Devices are then replaced by their new realizations, refer to Sect. 5.7.4. Using this approach, the result of Fig. 5.14d is obtained. The overall area utilization is far better than that of Fig. 5.14b.

5.7.3 Layout Hierarchy

It is important to preserve the hierarchy of the layout during migration. Hierarchical compaction has been first treated in [17] by formulating the problem as an LP one. The key is to represent all constraints contained in each cell (intracell constraints) only once in the constraint set even if it is instantiated multiple times. This is achieved by expressing edge locations inside hierarchical cells by their relative position with respect to their cell origin location rather than their absolute flattened position. For example, if an edge is located at a distance $x_{edge\_i}^{cell_k}$ from the cell origin, which in turn is placed at the absolute distance of x_{cell_k}, the edge location is expressed as $x_{cell\_k} \pm x_{edge\_i}^{cell_k}$ in all constraints. The cell location variable, x_{cell_k}, would then disappear from all intracell constraints, since it will be common to all edges. Therefore, intracell constraints reduce to the same set of constraints even if the cell is instantiated multiple times. This not only retains the layout hierarchy, but also reduces variable count in the simplex problem. However, if there exists multiple levels of hierarchy, the cell location variable, x_{cell_k}, would be in turn referenced to the origin of its parent cell. Therefore, an edge in a n-hierarchical structure is generally represented by

$$x_{cell\_i}^{top\_cell} \pm x_{cell\_i}^{level\_1} \ldots \pm x_{cell\_i}^{level\_n} \pm x_{edge\_i}. \tag{5.49}$$

Apparently, the expense of using this method is the increase in the number of variables in the linear constraint equations. In [32], the algorithm is further modified to trim down both the number of variables and the number of equations. Again, the presented multivariable graph-based algorithm would be a natural fit since it does not have any limitations on the number of variables in each constraint. While it is general enough so that no artificial techniques are needed to limit this number.

5.7.4 Device Replacement

During retargeting, some devices might have different structures and/or layers from one process to the other. For example, if the number of fingers of a given transistor needs to be changed, or if POLY resistors are not supported in the target process, simple layer mapping is not applicable. The only solution is to replace

the entire device with a new one from the target technology design kit. This process is facilitated by the adoption of parameterized cells or *Pcells* [33], where the new device is regenerated using a special Pcell script. A methodology for device replacement during migration has been presented in [34]. It is based on the ability to describe the entire hierarchical layout using multivariable linear programming constraints as described in Sect. 5.7.3.

5.7.5 Layout Parasitics

Layout parasitics have a significant impact on analog circuit performance, specially when it comes to matched nets. This effect is expected to be magnified in modern process technologies as parasitics control are getting worse. A recent publication has employed nonlinear optimization to constrain layout parasitics within predetermined bounds during retargeting [35]. This can be most useful for high-frequency and RF designs. It should be noted that if the source layout has been carefully designed with minimized and matched parasitics on sensitive nets, there is a quite large probability that the target layout also satisfy such constraints, specially if device dimensions do not change significantly due to migration. In this case, compaction objective functions that privilege closeness to the source layout [2] can greatly help keeping parasitics under control.

5.8 Examples

In this section, several compaction examples are given. First, results of the migration of two opamps: a two-stage Miller opamp and a folded-cascode one, are presented. Both have been migrated from a 0.35 μm process to a 0.18 μm one. New device sizes were calculated using the netlist migration tool reported in [10]. During layout migration, symmetry constraints were also considered. Using the multivariable graph-based method presented in Sect. 5.5, the CPU time of each test case was compared to that of the same migration using one of the well-known packages for the revised simplex method [36], both running on a 3.0-GHz, 512-MB RAM machine. Results are shown in Table 5.1. The table shows for each case, the number of location variables, the total number of constraints, the average single iteration time, and the total migration time for each algorithm. Since in the graph-based method, the graph is updated after each iteration to reflect new entering and leaving variables, the single iteration execution time changes from one iteration to the other depending on the new graph structure and the total number of loops in the graph. In this case, the average execution time of a single iteration is reported in the table. The table shows that the graph-based method is from two to four times faster than the simplex method in executing a single iteration.

Table 5.1 Comparison between graph-based and revised simplex methods

Case	Direction	# variables	# constraints	Avg. iteration time (ms)		Total time (sec)	
				Graph-based	Revised simplex	Graph-based	Revised simplex
Miller opamp	X	521	1,831	2.517	4.555	0.375	3.594
	Y	521	2,067	1.164	5.175	0.234	4.265
Folded-cascode	X	2,431	8,599	9.270	21.492	8.232	91.345
	Y	2,443	11,495	7.260	28.297	7.391	118.141

It should be noted that in both examples, the iteration CPU time of the graph-based method in the X-direction is higher than that of the Y-direction in spite of the fact that it has less number of constraints. This is due to the fact that the CPU time of one iteration depends not only on the problem size but also on the complexity of the coefficient constraint-graph, i.e., the number of loops that should be removed.

The total CPU time of the graph-based method is smaller than that reported for the revised simplex method by a factor of ten or more. This is due to two reasons: The first one is the much less CPU time needed for each iteration. The second reason is that starting from a good initial solution, as discussed in Sect. 5.5.2.3, has resulted in fewer number of iterations before a final optimal solution is reached.

For the Miller opamp, the source and target layouts are shown in Fig. 5.15a. The source layout area is 6,060 μm^2, while the target layout is 2,800 μm^2. For the folded-cascode, the source and target layouts are shown in Fig. 5.15b. The source layout area is 10,270 μm^2, while the target layout is 4,445 μm^2. Although LP has reached an optimal solution, there is still an empty area in the middle right part of the layout. This area appeared because of nonrelevant edge-constraints discussed in Sect. 5.6.

Another example of constraint problems is shown in Fig. 5.16. The source layout is shown in Fig. 5.16a. Migration is performed from a 130 nm process to a 65 nm one. After the first run, the obtained layout is shown in Fig. 5.16b. The relative size of the MOS capacitor device on the left side was greatly reduced with respect to devices at the top-right corner such that an empty space appeared on the top-left corner. However, due to the maintaining of edge-order, these devices could not be moved to the empty space. Similar problems can also be detected in the rest of the layout. After manual deletion of such constraints, the resulting layout of Fig. 5.16c has been obtained. A similar layout can also be obtained if another run of compaction is used in both directions. But in this case, a new set of constraints must be regenerated starting from the layout of Fig. 5.16b.

A final more complex example is the astable oscillator shown in Fig. 5.17. It contains several blocks, such as a bandgap reference, biasing cells, a digital decoder used for trimming, an amplifier, and a couple of comparators. The total number of devices is approximately 500. The migration was done from a 0.6 μm process to a 0.25 μm one. Table 5.2 shows the number of edges, the number of

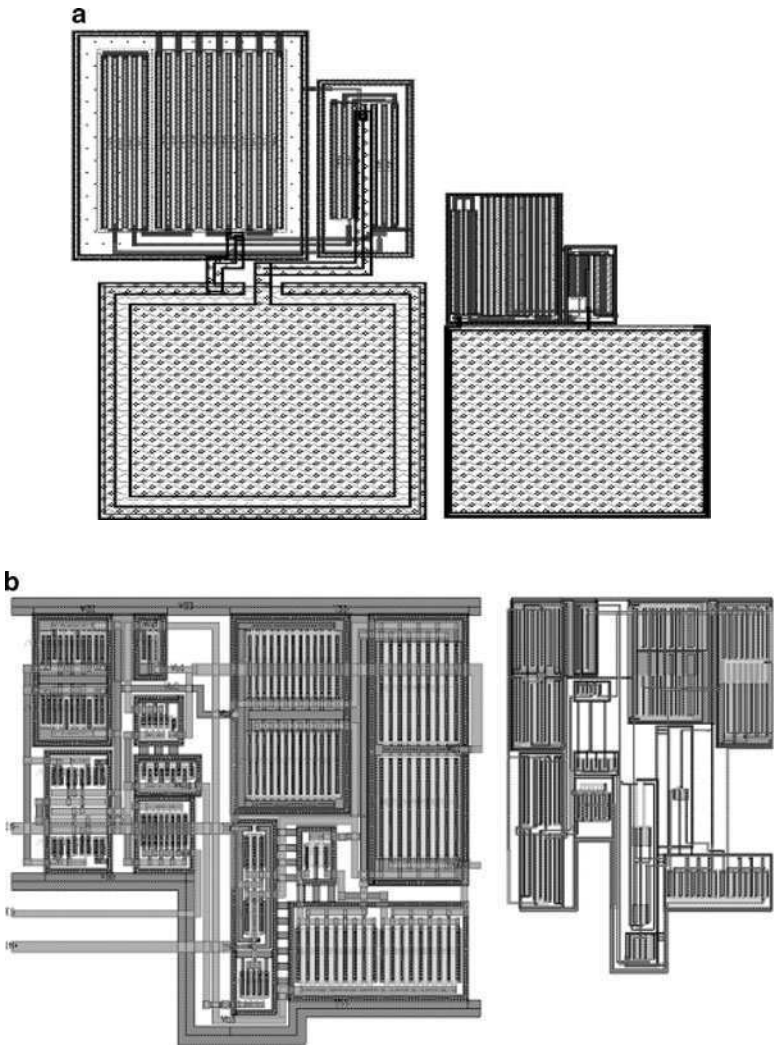


Fig. 5.15 Examples: (**a**) Miller opamp (**b**) folded-cascode opamp – *right* source and *left* target layouts

minimum-distance constraints, the number of symmetry and equal constraints, and the total number of iterations in both directions. As mentioned before, the number of minimum-distance constraints is by far greater than any other kind of constraints.In this example, some capacitors and resistors needed to be completely replaced by another device kinds as can be easily recognized from the layouts.

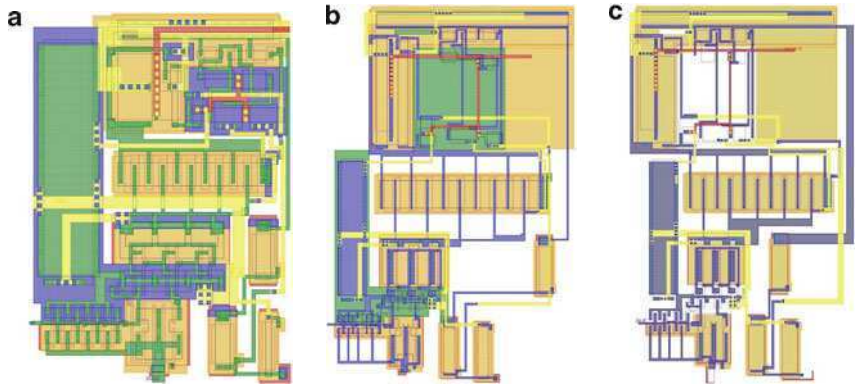


Fig. 5.16 VCO example: (**a**) source layout (**b**) migration with edge-order problems, and (**c**) migration after edge-order constraints deleted

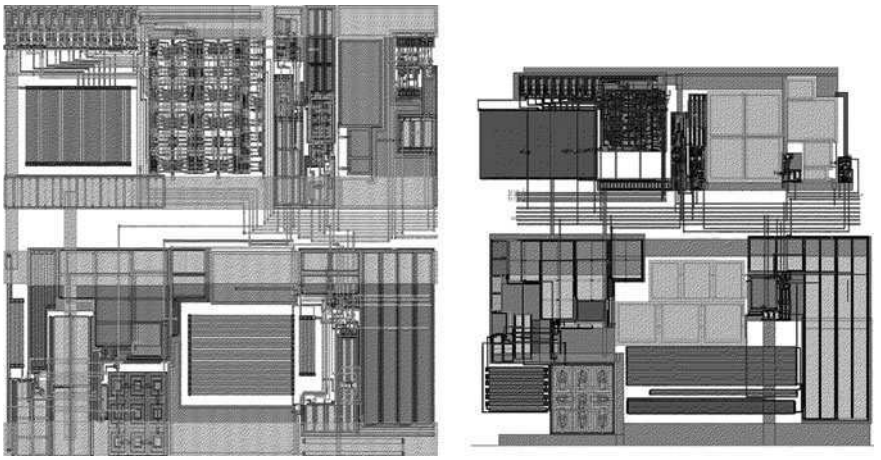


Fig. 5.17 Migration results for the oscillator migration

Table 5.2 Graph-based method compaction data of the example shown in Fig. 5.17

Direction	# edges	# min. constraints	# symmetry/equal constraints	# iterations
X	14,495	71,197	2,857	6,750
Y	14,469	75,494	2,676	6,540

5.9 Conclusion

In this chapter, it is shown that one of the most appropriate techniques for analog layout process migration is that based on layout compaction. With an appropriate objective function, compaction allows to preserve all physical design

knowledge embedded in the original layout. A novel multivariable graph-based simplex algorithm to be used in layout compaction was presented. It combines the efficiency of graph-based methods and the generality of linear programming ones. The algorithm is general enough to support any kind of complex multivariable constraint and any shape of linear optimization functions. Therefore, it is a natural fit for recently introduced symmetry, hierarchy and cell-swapping techniques based on linear programming. It is shown that the complexity of the proposed algorithm depends heavily on the source layout constraints and optimization function. In the limit case of simple minimum-distance constraints and a simple optimization function, it tends to the graph-based technique of linear complexity. In the opposite limit case of complex multi-variable constraints and complex optimization functions it tends to the complexity of the matrix-based simplex method.

Acknowledgements The authors would like to thank Mohamed S. Tawfik for initiating and supervising this project inside Mentor Graphics Egypt. Also, they would like to thank Hazem El-Tahawy for his continuous support. Finally, special thanks also go to the Chameleon-ART team: Walid Farouk, Heba Attwa, Sherif Hany, Manal Samy, Hoda El-Dawy, and Amr Ramadan.

References

1. J. Lakos. Technology retargeting for IC layout. In *Design Automation Conference, IEEE/ACM*, pages 460–465, 1997
2. J. Zhu, F. Fang, and Q. Tang. Calligrapher: A new layout-migration engine for hard intellectual property libraries. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 24(9):1347–1361, 2005
3. S. Bhattacharya, N. Jangkrajarng, and C.-J.R. Shi. Multilevel symmetry-constraint generation for retargeting large analog layouts. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 25(6):945–960, 2006
4. P.G. Drennan, M.L. Kniffin, and D.R. Locascio. Implications of proximity effects for analog design. In *Custom Integrated Circuits Conference. IEEE*, pages 169–176, Sept 2006
5. E. Malavasi, E. Charbon, E. Felt, and A. Sangiovanni-Vincentelli. Automation of IC layout with analog constraints. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 15(8):923–942, 1996
6. K. Francken and G. Gielen. Methodology for analog technology porting including performance tuning. In *Circuits and Systems, IEEE International Symposium on*, volume 1, pages 415–418, Jul 1999
7. S. Bhattacharya, N. Jangkrajarng, R. Hartono, and C.-J.R. Shi. Correct-by-construction layout-centric retargeting of large analog designs. In *Design Automation Conference, 41st*, pages 139–144, 2004
8. S. Hammouda, H. Said, M. Dessouky, M. Tawfik, Q. Nguyen, W. Badawy, H. Abbas, and H. Shahein. Chameleon ART: A non-optimization based analog design migration framework. In *Design Automation Conference, 43rd ACM/IEEE*, pages 885–888, 2006
9. Calibre®. Mentor Graphics Corporation. http://www.mentor.com
10. S. Hammouda, M. Dessouky, M. Tawfik, and W. Badawy. Analog IP migration using design knowledge extraction. In *Custom Integrated Circuits Conference*, pages 333–336, Oct 2004
11. Y.E. Cho. A subjective review of compaction. In *Design Automation. 22nd Conference on*, pages 396–404, June 1985
12. R. Okuda, T. Sato, H. Onodera, and K. Tamariu. An efficient algorithm for layout compaction problem with symmetry constraints. In *Computer-Aided Design, Digest of Technical Papers, IEEE International Conference on*, pages 148–151, Nov 1989

13. Y.E. Cho, A.J. Korenjak, and D.E. Stockton. FLOSS: An approach to automated layout for high-volume designs,. In *Design Automation Conference, IEEE/ACM*, page 138–141, 1988
14. Y. Liao and C.K. Wong. An algorithm to compact a VLSI symbolic layout with mixed constraints. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2:62–86, 1983
15. W.L. Schiele. Improved compaction by minimized length of wires. In *Proceedings of the 20th conference on Design Automation*, pages 121–127, Piscataway, NJ, 1983
16. D. Marple, M. Smulders, and H. Hegen. An efficient compactor for 45° layout. In *Proceedings of the 25th ACM/IEEE conference on Design Automation*, pages 396–402, Los Alamitos, CA, 1988
17. D. Marple. A hierarchy preserving hierarchical compactor. In *Design Automation Conference. Proceedings, 27th ACM/IEEE*, pages 375–381, Jun 1990
18. R.J. Vanderbei. *Linear Programming: Foundations and Extensions*, second edition. Kluwer, Dordecht, 2000
19. A. Onozawa. Layout compaction with attractive and repulsive constraints. In *Proceedings of the 27th ACM/IEEE conference on Design Automation*, pages 369–374, New York, 1990
20. L.-Y. Wang and Y.-T. Lai. Graph-theory-based simplex algorithm for VLSI layout spacing problems with multiple variable constraints. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20:967–979, 2001
21. L.-Y. Wang, Y.-T. Lai, B.D. Liu, and T.C. Chang. Layout compaction with minimized delay bound on timing critical paths. *Circuits and Systems, IEEE International Symposium on*, pages 1849–1852, 1993
22. R.C. Dorf. *Modern Control Systems*. Addison-Wesley, MA, 1995
23. P. Maidee and S. Choomchuay. Invisible edge and soft tied in compaction strategy. In *IEEE Asia-Pacific Conference on Circuits and Systems*, pages 133–136, Nov 1998
24. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT, MA, 1990
25. S.L. Lin and J. Allen. Minplex – a compactor that minimizes the bounding rectangle and individual rectangles in a layout. In *Design Automation. 23rd Conference on*, pages 123–130, June 1986
26. J.-F. Lee. A new framework of design rules for compaction of VLSI layouts. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 7(11):1195–1204, 1988
27. M.A. Riepe and K.A. Sakallah. The edge-based design rule model revisited. *ACM Transactions on Design Automation of Electronic Systems*, 3(3):463–486, 1998
28. S. Bhattacharya, S.H. Batterywala, Rajagopalan, H-K.T. Ma, and N.V. Shenoy. On efficient and robust constraint generation for practical layout legalization. In *Proceedings of the 9th international symposium on Quality Electronic Design*, pages 379–384, Washington, DC, USA, 2008. IEEE Computer Society
29. H. Shin, A.L. Sangiovanni-Vincentelli, and C.H. Sequin. "zone-refining" techniques for IC layout compaction. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 9(2):167–179, 1990
30. D.G. Luenberger and Y. Ye. *Linear and nonlinear programming*. Springer, Berlin, 2008
31. K. Okada, H. Onodera, and K. Tamaru. Compaction with shape optimization. In *Custom Integrated Circuits Conference. Proceedings of the IEEE*, pages 545–548, May 1994
32. J.-F. Lee and D.T. Tang. Himalayas-a hierarchical compaction system with a minimized constraint set. In *Computer-Aided Design. Digest of Technical Papers, IEEE/ACM International Conference on*, pages 150–157, Nov 1992
33. OpenAccess release 2.2. Silicon Integration Initiative. http//www.si2.org
34. S.H. Batterywala, S. Bhattacharya, S. Rajagopalan, H.-K.T. Ma, and N.V. Shenoy. Cell swapping based migration methodology for analog and custom layouts. In *Quality Electronic Design, 9th International Symposium on*, pages 450–455, March 2008
35. L. Zhang, N. Jangkrajarng, S. Bhattacharya, and C.-J.R. Shi. Parasitic-aware optimization and retargeting of analog layouts: A symbolic-template approach. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(5):791–802, 2008
36. lp_solve mixed integer linear programming solver. http://lpsolve.sourceforge.net/5.5/

Chapter 6
Closing the Gap Between Electrical and Physical Design: The Layout-Aware Solution

Rafael Castro-López, Elisenda Roca, and Francisco V. Fernández

Abstract Iterations between separate phases in any procedural design process, usually a by-product of unexpected (or, simply, very complex to consider) adverse effects, clearly play against any time-to-market requirements. In analog integrated circuit (IC) design, going back and forth between electrical and physical synthesis to counterbalance layout-induced performance degradations needs to be thus avoided as much as possible. One possible solution involves the integration of the traditionally separated electrical and physical synthesis phases, by including layout-induced effects, in the form of layout parasitics, right into the electrical synthesis phase, in what has been called parasitic-aware synthesis. This solution, as such, is not yet complete since there are geometric requirements (minimization of the occupied area or fulfillment of certain layout aspect ratio, among others), whose effects on the resulting parasitics are not usually considered during electrical synthesis. In this chapter, a layout-aware solution that tackles both geometric and parasitic-aware electrical synthesis is proposed. This technique uses a combination of simulation-based optimization, procedural layout generation, exhaustive geometric evaluation algorithms, and several mechanisms for parasitic estimation. Thanks to the nature of this combination, the solution benefits from, and also fosters, reuse of analog intellectual property (IP) blocks. Several detailed design examples are provided.

6.1 Introduction

The CMOS semiconductor industry has continuously evolved and prospered since the early 1970s. The ever-shrinking minimum feature size triggered a revolution in the electronic industry, from ASICs (application-specific ICs), to SoCs (Systems-on-a-Chip), SiPs (Systems-in-a-Package), and NoCs (Networks-on-a-Chip). A critical

R. Castro-López (✉)
Instituto de Microelectrónica de Sevilla, IMSE-CNM-CSIC and University of Seville, Spain
e-mail: castro@imse-cnm.csic.es

H.E. Graeb (ed.), *Analog Layout Synthesis: A Survey of Topological Approaches*,
DOI 10.1007/978-1-4419-6932-3_6, © Springer Science+Business Media, LLC 2011

design productivity lag, however, has been reported [1]: with a 58% yearly growth in IC complexity considerably surpassing the 21% yearly increase in productivity, design cost is increasing rapidly. Taking into account the ever-demanding time-to-market pressures, this picture is clearly worrisome. For analog and mixed-signal (AMS) design, the situation is even worse, one of the most significant causes being the scarcity of commercial CAD tools and methodologies to support the analog design efficiently. In this scenario, productivity should be boosted to double every year to bridge the gap. To achieve this goal, several research directions have been suggested [1], among others: (1) increasing the fraction of the design coming from reuse-based design practices, (2) improving hierarchical synthesis methods so that subtle analog design knowledge is more efficiently managed, and (3) avoiding iterations between separate design stages. However, the design of analog integrated circuits is, in many ways, a very intricate process that demands to be systematized as long as it is both possible and productive. Here, systematic means that the design process is organized in such a way that it is efficient with respect to the available resources (like CPU time); many, if not all, parts of the process could be, if wanted, properly automated; and it provides equal, if not better, results when compared to a traditional, nonsystematic, handcrafted design process.

The research and results described in this chapter focus on the three directions mentioned above, with special emphasis on the third one, which will be undertaken by minimizing the iterations between electrical synthesis (also known as sizing and here understood as the process of mapping performance specifications into device-level characteristics, such as transistor sizes and biasing conditions) and physical synthesis (or layout generation). At the same time, this chapter also deals with the systematization of both these electrical and physical parts of the design process, by mixing them up and, in this way, getting to a design process that avoids time-consuming iterations. In doing so, it is possible to completely implement those three meanings of systematization.

To comprehend the goals to be achieved, it is necessary to understand that the quality requirements in analog and mixed-signal (AMS) design, as well as productivity levels in the semiconductor industry involve many aspects [1]. Quality means that the performance of the fabricated circuit is guaranteed even in the presence of layout-induced parasitics. This goal is usually achieved through time-consuming and unsystematic iterations between the electrical and physical design phases. Failing to attain the goal may eventually lead to product-to-market failure, but the iterations are clearly a pitfall in designer's productivity. In this sense, it is important to note that the origin of this iterations is the disconnection in the design flow that exists between electrical and physical design. Actually, an important aspect of this issue is how to evaluate layout-induced parasitics early in the flow: in traditional design, overestimation results in wasted power and area, while underestimation may lead to fatal performance degradation. Another very important aspect of quality is the effective use of silicon area because this is paramount to the final production cost. This goal can be assured by carefully optimizing the design from the point of view of geometry by, for instance, improving layout regularity and floorplanning or attaining certain aspect ratios for the layout components.

These two quality aspects, area and parasitics, are nonetheless intimately related: to optimize the design from point of view of the geometric unavoidably affects the robustness of the design against parasitic-induced effects, and vice versa [2].

Beyond traditional analog circuit design techniques, the majority of the reported solutions to implement more systematic and automated techniques cover only one of the both problems, and just a few contributions have tried to address both of them simultaneously. Another important aspect concerns the phase of the design cycle where the solutions are applied: either after circuit sizing or concurrently with it.

The synthesis methodology presented in this chapter, so-called layout-aware sizing, aims at concurrently solving the geometrical and parasitic problems by bringing layout-related data into the very sizing process. Circuit sizing can then be carried out with enough information on layout-induced degradations (parasitics) as well as with a detailed description of the geometry of the eventually implemented layout. In this way, circuit sizing ensures a solution that is robust enough against layout-induced degradation effects and that fulfills a number of user-defined geometric goals, with area minimization being the most important. The salient features of the proposed solution are:

1. Thanks to the simulation-based approach, the methodology is very flexible and general since many different types of circuits can be synthesized. Also, the use of an electrical simulator provides a high level of accuracy in the simulation of the circuit performances.
2. The global optimization techniques ensures that high-performance solutions (that meet very demanding specifications) are attained.
3. Because accurate parasitics estimates are incorporated right into the electrical synthesis phase, the performance of the design solutions is guaranteed.
4. The execution times are kept within reasonable limits thanks to the efficient optimization-based and template-based layout generation techniques.
5. Minimization of area during electrical synthesis is done in a much more realistic manner because, as opposed to traditional electrical design, every detail on the layout implementation (e.g., routing, guard-rings, block separation, etc.) is taken into account.

To the best of the authors knowledge, previously mentioned reported approaches only meet some of the previous features. It is also worth noting that the methodology presented here fosters reuse-based design practices for analog and mixed-signal IP blocks, another featured direction toward new IC design methodologies. AMS design expertise regarding electrical and physical IC design can be stored by using techniques explained next. Thus, this expertise can be swiftly and efficiently reused for many different design scenarios.

The layout-aware synthesis methodology here described is composed of two sizing techniques, namely the parasitic-aware and geometrically constrained sizing techniques. The latter technique concerns the inclusion of layout knowledge in the sizing process to obtain a solution for which the area and shape of the eventually implemented layout are optimized. Such a goal is accomplished by finding, during the sizing process, the values of geometric parameters (e.g., number of folds

of MOS transistors) that yield optimal geometric features. This optimization can be defined either as a restriction on some geometric aspects of the layout (e.g., a predefined aspect ratio, or a maximum width or height of the whole circuit layout) or as a design objective (i.e., area minimization). Two aspects should be here carefully considered. First, adding new variables to the sizing process (i.e., geometric variables in this case) must not simply consist in extending the design space because it would make the exploration of such a space exponentially more complex. Second, to be able to include layout details into the sizing process, layout generation must be rapid enough so that retrieving these details does not overly slow down the sizing process. In parasitic-aware sizing, the values of layout parasitics are computed interactively during sizing, by using specific layout information (e.g., the possible implementation style of a group of MOS transistors) and actual device sizes. The eventually obtained sizing yields a performance that is also robust when considering these layout-induced parasitic effects. As previously noted, these two techniques have been carefully linked as different geometric variables may give rise to various, different layout-induced effects (e.g., different foldings change the junction capacitances of an MOS transistor).

The structure of this chapter is as follows. Following the review of previous work in Sect. 6.2, Sect. 6.3 describes the circuit sizing and layout generation techniques used as foundations to develop the layout-aware synthesis of AMS circuits. Section 6.4 explains the geometrically constrained sizing technique, while Sect. 6.5 completes the layout-aware sizing methodology with the parasitic-aware technique. Several design examples are provided in both sections. Conclusions are drawn in Sect. 6.6.

6.2 Previous Work

6.2.1 Circuit Sizing and Layout Generation

Synthesis can be carried out by following two different approaches, the first based on knowledge, the second founded on optimization.

The basic idea behind knowledge-based synthesis is to use a predefined design plan (in the form of design equations, design heuristic strategies, or both) to find and combine the elements such that the set of requirements (for sizing or layout) are met. The underlying principle is to capture the expertise of a designer so that an optimum solution can be reached.

In knowledge-based sizing, design equations and heuristics are formulated in such a way that, given the required performance characteristics, the component's characteristics can be calculated (e.g., [3]). Although it reaches solutions quickly, this approach suffers from several drawbacks: limited accuracy due to the use of simple equations, large preparatory time/effort and difficult migration to different technologies.

Used for layout synthesis the intended captured knowledge refers to the wide variety of techniques (from placement strategies, intended to improve device matching and minimize the layout area, to routing techniques, used to minimize the loading effects) that expert layout designers use to improve the quality of the layout. There are two types of knowledge-based layout synthesis approaches, namely rule-based and template-based approaches. Rule-based approaches store the layout knowledge in a customizable rule set (defining what a "good" analog layout), which are to be obeyed by whatever layout placement and routing algorithms are to be used. However, most common knowledge-based approaches are template-based tools [4–9]. The underlying idea with template-based tools is to capture the layout designer's expertise in a pattern that specifies all necessary component-to-component and component-to-wiring spatial relationships, as well as analog-specific constraints such as symmetry, device matching, and parasitic minimization.

In optimization-based synthesis, on the other hand, the problem is translated into function optimization problems that can be solved through iterative numerical methods [10–21]. Although the optimization may be single or multi-objective (i.e., using one or multiple functions that are minimized and/or maximized simultaneously), the main idea is that the quality of the circuit performances are iteratively quantified and compared with the performance specifications. This quantification requires an evaluation that can be done by using equations (derived either manually or using symbolic analyzers [11–16]), or by using a simulation tool [17–21]. Whereas the simplicity of the equations may compromise the accuracy of the solution, using a simulation tool improves the accuracy but the running time is typically much larger than when using equations. However, the simulation-based optimization is conceptually more general since it can be used with many different type of circuits.

Regarding the optimization technique, two approximations are considered: statistical [16, 20, 21] and deterministic [17]. The main advantage of the statistical techniques over the deterministic ones is their capability to escape from local minima, thanks to a nonzero probability of accepting movements that may increase the cost function. The price to pay is, however, a larger computational cost.

In optimization-based layout generation [22–24], the cost function typically considers some design aspects such as area and net length, while penalizing violation of analog design constraints, such as device mismatch, or crosstalk. The main benefit is their generality since they can be applied to any AMS circuit. The drawbacks are the complexity of the optimization problem, the relatively low quality of solutions, the difficulty of the cost function setup, and the long turnaround times.

6.2.2 Previous Approaches to Layout-Aware Sizing

In this section, we analyze different reported approaches to layout-aware circuit sizing. This analysis attends to several aspects, such as the engine used for sizing (knowledge or optimization-based), the kind of evaluation of the circuits performances (equations or simulation), the estimation method for parasitics (i.e., the type

Table 6.1 Reported approaches for layout-aware sizing

Ref.	Sizing engine	Performance evaluation	Estimation method	Layout generation	Geometric constraints
[16]	Performance models & genetic algorithms	Fitted functions	2.5-D analytical-geometrical	Yes	Equations
[25]	Design plans	Equations	Analytical-geometrical	No	Linear programming
[26]	Simulation-based non-linear opt.	Numerical simulation (SPICE)	Analytical-geometrical	Yes	No
[27]	Simulation-based opt.	Numerical simulation (NG-SPICE)	Analytical & look-up tables	No	No
[28]	Simulation-based opt.	Symbolic analysis	Off-the-shelf extractor	Yes	No
[29]	Equation-based opt.	Matrix nodal analysis	Analytical models	No	No
This work (1) [30]	Simulation-based opt.	Numerical simulation (HSPICE)	Analytical & layout-sampling	No	Slicing tree
This work (2) [30]	Simulation-based opt.	Numerical simulation (HSPICE)	Geometric & 3-D analytical and geometric	Yes	Slicing tree

and accuracy of the extraction process), the inclusion of geometry-aware information, and the need for layout generation in the approach. The analysis is summarized in Table 6.1, which includes also the solution described in this chapter.

In [16], the sizing engine uses simple performance models and evolutionary algorithms, analytical–geometrical models for parasitics, and procedural layout generation. Geometrical concerns are addressed by using coarse equations describing the geometries of the layout components that are required for each solution. The information on extracted parasitics is, however, very limited.

A knowledge-based sizing approach that uses design plans is described in [25]. Parasitics are evaluated only for a relatively low number of iterations within the sizing process. The estimation method for parasitics is based on analytical–geometrical models. Geometrical concerns are considered by means of a linear programming technique: the layout is optimized at the slice level and a simplex algorithm is applied to a vertically (or, correspondingly, horizontally) stacked set of horizontally (vertically) arranged devices (a building block), called groups. Since this approach uses design plans, the accuracy of evaluated solutions is compromised. Moreover, even with the most accurate models for parasitic estimation, the resulting performances may wrong when added to the approximate models used in the design plans. Another drawback of this approach is that geometric aspects are not really considered as a part of the design space exploration during the sizing process. This means

that area is not an objective that is actually and comprehensively minimized during sizing (it is only taken into account when near the optimum, i.e., during local optimization). Besides, initial heuristic estimates are used to solve the problem, which makes the solution highly dependent on starting guesses.

The solution in [26] consists in a first global sizing phase with no information on layout followed by a detailed sizing phase, which generates a procedural layout at each iteration. Therefore, parasitics are only considered for fine-tuning. Geometric concerns are limited to the minimization of area through the number of folds in MOS transistors.

The layout-aware technique presented in [27] uses simulation and optimization combined with a parasitic estimation method based on analytical models and look-up tables (the latter used for routing wires). The main drawback of this solution is that no geometry-aware information is taken into account during the sizing process.

The solution in [28] uses layout with templates and commercial extractors. Parasitics are incorporated into performance models by using symbolic analysis. Although gain in sizing time is reported to be approximately 20% when compared with traditional simulation-based techniques, the solution is limited to small-signal performances. Besides, no geometric concerns are included.

The work in [29] uses equation-based models to evaluate circuit performances and analytical models to estimate parasitics. Again, the use of equations limits the accuracy of evaluated performances (added to the relative imprecision of parasitic estimates). Also, geometry-related aspects are not considered.

6.3 Selected Approach to Layout-Aware Sizing

6.3.1 Sizing

Although knowledge-based sizing does reach solutions quickly, provided that design plans have been already derived, this approach suffers from several drawbacks. The most important one is that the quality of the solutions in terms of both accuracy and robustness is not acceptable since the very concept of knowledge-based sizing forces the design equations to be simple, thereby resulting in large deviations of the real performance from the predicted one (several hundred percent in worst-case scenarios). Other drawbacks are the large preparatory time/effort required to develop design plans or design equations, the difficulty in using them in a different technology, and the ad hoc nature of the approach itself. Optimization-based sizing circumvents the need for a detailed design plan. Even though there is a clear gain in running times when using equations, this type of performance evaluation method may, however, compromise the accuracy of the solution. With optimization-based sizing using simulation, better accuracy can be attained as long as accurate simulation models are used, but the running times are typically much longer than sizing with equations. However, optimization-based sizing is conceptually more general, the simulator (i.e., the performance evaluator) being the component that determines the applicability to different types of circuits and topologies.

For these reasons, a simulator-in-the-loop, optimization-based engine has been chosen to perform circuit sizing [20]. This engine features some characteristics in the generation and acceptance of movements through the design parameter space that allow to drastically reduce the computational cost, such as: preliminary exploration of the design space using a coarse multidimensional grid to determine the best regions for further exploration, adaptive control of the temperature in the simulated annealing statistical techniques, synchronization of movement amplitude in parameter space with the temperature, among others. An outstanding feature is the capability to incorporate design knowledge to the sizing procedures, which can be done by making use of powerful tools like embeddable C-based programs. This is very important because (as it is shown below) this capability makes it possible to introduce layout-related aspects directly into the sizing process.

Then, the optimization problem is mathematically stated as:

$$\text{Minimize} \quad y_{oi}(\mathbf{x}), \qquad 1 \leq i \leq P, \tag{6.1}$$

$$\text{Subject to} \quad y_{rj}(\mathbf{x}) \geq Y_{rj} \quad \text{and/or} \quad y_{rj}(\mathbf{x}) \leq Y_{rj}, \qquad 1 \leq j \leq R, \tag{6.2}$$

where $y_{oi}(\mathbf{x})$ stands for the value of the i-th design objective (e.g., minimize power consumption); $y_{rj}(\mathbf{x})$ is the value of the j-th design constraint (e.g., phase margin larger than 45°); Y_{rj} is the targeted value of such a design specification; and $\mathbf{x}$ is the vector of design variables (e.g., transistor sizes, capacitor values, bias currents, etc.).

In the context of circuit design, there is an important difference between a design objective and a constraint. While the former are meant to improve the design, the latter are set to defined what is a valid or feasible design. An example of design objective is power consumption to be minimized; an example of constraint is to have a phase margin above 45°. It is also important to note that it is the choice of the designer to state both constraints and objectives (sometimes, what is defined as constraint can be used as objective, and vice versa).

In the work presented here, a single-objective optimization engine is used, so a cost function is used. This cost function is defined according to the feasibility of the point of the design space that is being evaluated during the optimization. If the point does not satisfy any of the design constraints, the cost function is to be defined as:

$$\psi(x) = \max[-w_j \log(y_{rj}/Y_{rj})], \tag{6.3}$$

where w_j is the weight associated to the jth constraint.

For those points of the feasible design space, the cost function is defined as follows:

$$\psi(x) = \Phi(y_{oi}) = -\sum_i -w_i \log(|y_{oi}|), \tag{6.4}$$

where w_i is the weight associated with the ith design objective used to prioritize the fulfillment of one or more design objectives over others.

The optimization engine explores the design spaces in two phases [20]. In the first one, the best regions for further exploration are determined through a simulated

annealing technique combined with an adaptive temperature control. In the second phase, a deterministic technique Powell's method [31] performs the local optimization of the design.

As evaluator of the circuit performances, any transistor-level simulator (e.g., HSPICE) can be used. In this way, the layout-aware sizing methodology described here can be applied to any analog circuit that can be efficiently simulated at the device level.

6.3.2 Layout Generation and Parasitic Extraction

The traditional design flow in AMS circuit design regarding layout-induced parasitics presents several drawbacks. The trial-and-error approach to counteract the adverse impact of parasitics involves an unsystematic method to correct the circuit sizing, its layout, or both. Oftentimes, the resulting degradation of the circuit performance may be due not to a single parasitic but to the combined effect of several parasitics. Moreover, the number of extracted parasitics (which depend on the complexity of the circuit and the accuracy of the extraction tool) makes this correction method even more time-consuming and error-prone. All in all, the number of required iterations to ease the impact of parasitics can be quite significative for typical analog circuits.

On the other hand, accurate extraction of parasitics requires knowing the circuit layout in detail. If parasitics are to be estimated during the sizing process, this layout knowledge must be generated at each iteration of the sizing process. Therefore, any method used to obtain this information must be fast enough to prevent circuit sizing from taking prohibitively long. A feasible way to attain such information is to generate the layout at each iteration. In this regard, optimization-based layout generation is currently too slow to be called in the loop of an automated parasitic-aware sizing process.

In this sense, template-based layout generation is a more suitable solution for several reasons[1]: (1) the time required for layout generation (a few seconds [5]) is considerably smaller than those required by optimization-based approaches; (2) layout templates are parameterized structures that contain all information on the final circuit layout implementation, which is essential to accurately estimate parasitics and, therefore, minimize the number of iterations between sizing and layout [8,9,32]; (3) layout templates are very efficient at encapsulating design expertise, both for placement and for routing.[2]

[1] Layout templates, which are to be devised by following expert guidelines for analog layout, are, however, relatively costly to generate. Providing that a library of basic building blocks (such as transistors, resistors, and capacitors) is available, developing a layout template takes $\times 1.5 - \times 2$ the time it takes to manually create the layout for the same block.

[2] For instance, device symmetries can be coded right into the building block so that any instance of the template ensures that pertinent symmetries are always kept.

Despite these advantages, layout templates may not result the best solution in terms of generality, since they lack flexibility (in terms of placement and routing adaptation) for certain sizes of the circuit devices. Nevertheless, the geometrically constrained sizing technique described below together with a careful design of layout templates contribute to palliate their flexibility problems to a large extent. Our template-based approach has been implemented [5] using the Cadence's PCELLS technology [33] and SKILL programming [34].

6.3.3 Putting It All Together

Figure 6.1 shows the flow diagram of the layout-aware sizing methodology. Its core, the optimization engine based on simulation explained above, features the means to add relevant designers' expertise to such iterative optimization process. As, in general, the design space of any circuit is a multidimensional space defined by all the design parameters (e.g., physical parameters of transistors, resistors, and capacitors), adding such expertise is necessary to bind the exploration of the circuit design space to only those regions yielding more suitable solutions, thereby improving the efficiency of the procedure. By making use of powerful tools such as embeddable C-based executables, it is possible to incorporate valuable design expertise in the form of constraint-satisfaction equations. This capability enables carrying out the floorplan-sizing task at each iteration of the optimization process. The C++

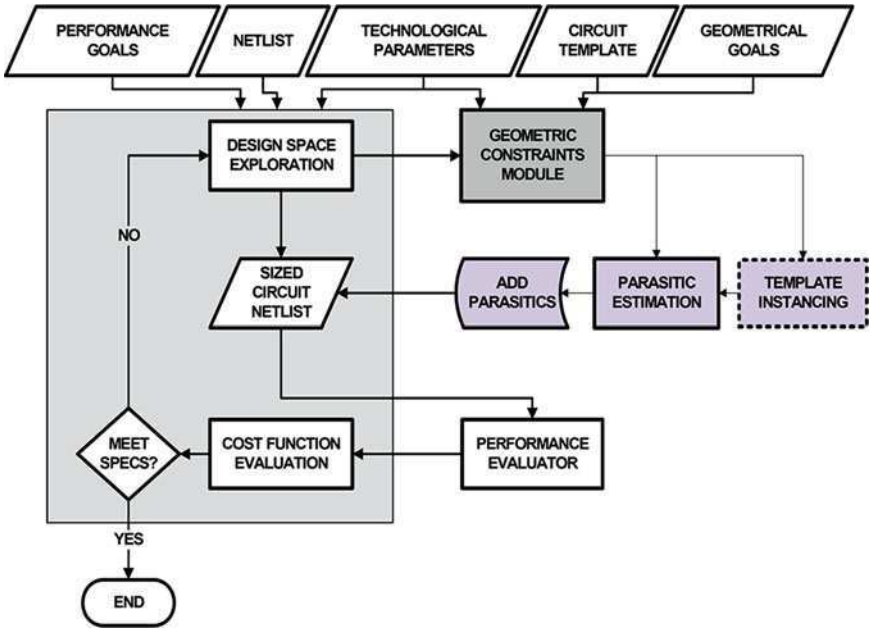


Fig. 6.1 Block diagram of the layout-aware sizing methodology ([30], ©IEEE 2008)

program labeled geometric constraints module (hereafter referred as the GC module) in Fig. 6.1 has been created to perform the floorplan-sizing task following the technique explained above.

The data and information required in this flow are:

1. The circuit description (netlist) and the electrical performance to be posed as goals of the optimization process.
2. Circuit template description: binary slicing tree representation of the layout template the GC module works with.
3. Geometric goals: user-specified objectives concerning the geometric characteristics of the eventually generated circuit layout
4. Fabrication process data, mainly layout design rules and process parameters.

The flow of optimization at any given iteration proceeds as follows:

1. A new vector for the design variables (within a prespecified range) is selected. At the beginning of the optimization, this selection is done randomly. Otherwise, the selection is done according to the optimization algorithm that is in place (e.g., following the simulated annealing optimization algorithm).
2. The vector of design variables is passed on to the GC module, which calculates all possible combination of geometric parameters rendering all possible layout styles for all of the circuit components. According to the required geometric objectives (e.g., aspect ratio, maximum width, etc.), the GC module outputs the combination of geometric parameters that attain those objectives while both area and area loss are minimal.
3. With all necessary information to generate the layout, an instance[3] of it is extracted and the resulting parasitics are added to the circuit.
4. The circuit performances are evaluated in the presence of the extracted parasitics.
5. Electrical performances and geometric features are used in the cost function that controls the optimization procedure.

6.4 Geometrically Constrained Sizing

The goal of geometrically constrained sizing is to help the designer in finding the best use of the available silicon area. This is done by retrieving and using correctly and accurately geometry-related information during the electrical design process. Beyond the obvious benefits that this has in the physical implementation from the geometry perspective (attaining a layout aspect ratio and improving the layout quality and area usage efficiency), an important benefit is that parasitic estimates can be obtained and used right in the very sizing process.

[3] An instance is the result generating an actual layout from a template, with a particular set of values for the template parameters.

6.4.1 Floorplan Sizing

The geometry optimization problem for integrated circuits is known as floorplan-sizing problem. Given a collection of device sizes (e.g., width and length of MOS transistors) for a given analog circuit and, since each device can be laid out in multiple ways, the floorplan-sizing problem consists in finding out two things: the placement (i.e., how the devices are place in relation to each other) and the geometric parameter (GP) values (e.g., the number of fingers of a MOS transistor) of all components, such that some function $\Psi(W, H)$ of the whole circuit layout width, W, and height, H, is minimized. Note that the placement is already established by the layout template that is being used, so the only problem is finding out the appropriate values of the geometric parameters.

In solving this problem, we thus seek to find the width and height of the layout of each circuit component and, correspondingly, the values of its GPs that minimize Ψ. An example of such a function is the total occupied area $\Psi(W, H) = W \cdot H$. Another interesting function is the relative amount of area loss with respect to the total occupied area. This loss results from the fixed, prestored placement, and routing of components in template-based layouts [5]. The area loss figure gives an idea of how further could the layout be compacted if other device placement is used instead. This figure is computed as the ratio between the total area and the area that is not used by any block and routing wire, and that is not used with the design rules (e.g., n-well spacing). The minimization of the area loss figure in the optimization process provides an additional way to improve the efficiency (in terms of area usage) and the flexibility of template-based layout solutions.

In our approach, the slicing style has been used to specify the layout floorplan [35]. A slicing floorplan is obtained when the layout components are arranged such that the layout area is recursively divided into horizontal and/or vertical slices. In a slicing floorplan, a slice is a combination of two or more components, either building blocks or further slices. Although nonslicing floorplans are a more general representation that can describe all kinds of tile packing, slicing floorplans have important advantages over nonslicing, namely:

- Placement can be more easily specified by the relative positions of the layout tiles, since the hierarchy of slicing structures is better defined.
- It yields more compact layout instances.
- It also allows evaluating other characteristics of the circuit layout, such as routing more easily.
- It eases geometrically constrained sizing.

In our work, we have used rooted binary trees to represent the hierarchical structure of the slices in a slicing floorplan. These trees, also known as slicing trees, have nodes that can be classified as leaf nodes and nonleaf nodes. In the tree, there are m nonleaf nodes and $n = m + 1$ leaf nodes. Each node corresponds to a block component (each of one implementing one or more circuit devices) of the layout. For instance, the slicing tree for the fully differential amplifier in Fig. 6.3 is shown

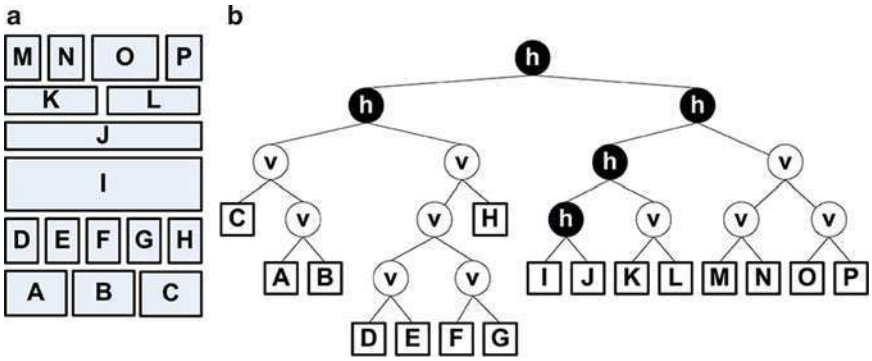


Fig. 6.2 (**a**) Representation of the floorplan and (**b**) slicing tree for the circuit in Fig. 6.3 ([30], ©IEEE 2008)

Table 6.2 Slicing tree and devices of the circuit in Fig. 6.3

Slice	Tile	Devices	Geometric parameter (GP)	Building block
1	A	R_{z1}	Rz_m (no. of fingers)	Folded resistor
	B	M_3, M_4, M_{10}, and M_{11}	$m3$ and $m4$ (no. of fingers)	Cascode structure
	C	R_{z2}	Rz_m (no. of fingers)	Folded resistor
2	D	C_{c1}	Xcc (horizontal dim.)	Unit capacitor
	E	M_{14}	$m14$ (no. of fingers)	Folded transistor
	F	M_1, M_2	$m1$ (no. of fingers)	Differential pair
	G	M_{15}	$m14$ (no. of fingers)	Folded transistor
	H	C_{c2}	Xcc (horizontal dim.)	Unit capacitor
3	I	M_6–M_9	$m6$ (no. of fingers)	Cascode structure
4	J	M_5	$m5$ (no. of fingers)	Current mirror
		M_{12}, M_{13}	$m12$ (no. of fingers)	
5	K	R_{CM1}	Rcm_m (no. of fingers)	Folded resistor
	L	R_{CM2}	Rcm_m (no. of fingers)	Folded resistor
6	M	M_{5c}	$m5c$ (no. of fingers)	Folded transistor
	N	M_{3c}	$m3c$ (no. of fingers)	Folded transistor
	O	M_{1c}, M_{2c}	$m1c$ (no. of fingers)	Differential pair
	P	M_{4c}	$m3c$ (no. of fingers)	Folded transistor

in Fig. 6.2. The floorplan in Fig. 6.2a has six slices with one to five horizontally distributed components. The corresponding slicing tree (which in this case has minimal depth) is shown in Fig. 6.2b.

The correspondence between the leaf cells and the devices of the opamp is explained in Table 6.2. Note that each leaf node has several possible shapes and, therefore, several possible values of the pair $\{width, height\}$. These pairs come from varying one or more GP of the devices in the leaf node (see, for example, Table 6.2).

The collection of Pareto-optimal[4] pairs $\{width, height\}$ for a leaf node forms its *shape function*; from the shape function it is possible to retrieve the height value that corresponds to a certain with value, and vice versa.

6.4.2 Proposed Approach

For AMS circuits, there are two different approaches to solve the floorplan-sizing problem. Both of them start from a predefined floorplan, and, therefore, the relative block placement is fixed. The first approach is based on the Stockmeyer's algorithm [36], widely cited in the digital arena. The second approach is based on the formulation of the floorplan sizing as a linear programming problem and the application of the simplex method to solve it. An implementation of this second approach is reported in [25], already discussed in Sect. 6.2.

Reported applications of the Stockmeyer's algorithm on analog or mixed-signal design tackle floorplan sizing only after circuit sizing [4, 37]. Therefore, neither layout optimization nor layout-induced parasitics are concurrently considered with sizing, in contrast with the solutions presented in this chapter.

To solve the issues that previous approaches have, such as separate electrical and floorplan sizing, the solution we present in this chapter pursues the following three goals to be undertaken during the sizing process:

1. Minimize both the occupied area and the ratio of unused silicon area (i.e., the area loss).
2. Attain a complete and detailed description of the layout geometry (with the values of all geometric parameters) to evaluate correctly all layout-induced parasitics and to have an accurate measure of the area that the layout occupies.
3. Provide solutions featuring user-specified constraints on the geometry, such as the aspect ratio or the maximum width and/or height of the layout.

A key component is the GC module, which performs the floorplan-sizing task based on a modified Stockmeyer's algorithm. This algorithm considers for each component two pair of width–height values that comes from taking only two possible shapes of the component (the second one being the 90°-rotated version of the first one). That is, the shape function of the component is formed by the pair of values $\{(h, w), (w, h)\}$, with h and w being the height and width of the component, respectively. Applying this algorithm to the floorplan sizing problem in analog circuits means that each building block has a list of heights and widths coming from different styles of implementation (e.g., common-centroid, interdigitized, etc.) and/or the different values of its GPs (e.g., different number of fingers or strips).

[4] A Pareto-optimal element is an element that has lower width and/or height than the rest of elements. That is, it is not possible to find any other element that has a lower value of both width and height than the Pareto-optimal element.

The GC module in Fig. 6.1 works in two phases. In the first phase, the bottom-up phase, the slicing tree (see, for instance, Fig. 6.2b) is processed bottom-up, beginning by associating a list with each leaf node of the tree (sorted according to the rules $h_i > h_{i+1}$ and $w_i < w_{i+1}$) that represents the building block shape function. To do this, there is a database of generators that provide the shape function given the type of leaf node (e.g., a folded transistor, an MOS differential pair, an MOS cascode group, a current mirror group, a capacitor array, or a folded resistor), the size(s) of the device(s) and the fabrication process. Note that the pcells and templates that are used to implement each type of leaf node were made such that they reflect several choices from the electrical and geometrical point of view, with parameters that make them flexible enough to ensure device matching, shielding, and reliability[5] [5].

Once every leaf node has been processed, the GC module combines them into vertical, v, and horizontal, h, nonleaf nodes, according to the slicing tree that has been defined for the circuit layout. For each nonleaf node, a list of s pairs, $\{(h_1, w_1), ..., (h_s, w_s)\}$, is generated. This combination follows the rules of Stockmeyer's algorithm. The list of pairs is built with the following properties:

1. $s \leq \prod_{i=1}^{L(v)} l_i$, where $L(v)$ is the number of leaf nodes of the subtree rooted at v and l_i is the cardinality of the shape function of leaf node i.
2. Pair (h_i, w_i) is kept in the list unless there is another (h_j, w_j) that is strictly better (lower) in the h or w dimension (or both) and is not worse than (h_i, w_i) in either dimension (remember that this is the definition of Pareto optimality).
3. Pairs are sorted according to the rules $h_i > h_{i+1}$ and $w_i < w_{i+1}$.

Additional information from the layout template (such as the physical separation between slices) is provided to help composing the top-most shape function accurately. By recursively applying the algorithm from bottom-up the slicing tree, the first phase ends with the associated list or global shape function of the overall slicing tree. As a side note, it is worth mentioning that the complexity of this shape function algorithm is $O(dl_T)$ with d being the depth of the slicing tree and l_T the sum of the cardinalities of the shape functions of the leaf nodes in the slicing tree.

Once the global shape function is attained, any function $\Psi(W, H)$ can be calculated. Actually, the GC module's main output is a matrix, called floorplan-sizing matrix. The first two columns of this matrix are the width, W, and height, H, of every point in the global shape function. The next columns account for several $\Psi(W, H)$-functions, such as the area and the aspect ratio. The area loss is calculated as the difference between $W \times H$ (the occupied area) and the sum of all $W_i \times H_i$ products (with i representing every leaf-node) plus the area required by the routing wires. For the sake of illustration, Fig. 6.4a shows the shape function and the aspect ratio of the opamp in Figs. 6.2 and 6.3, which in this case contains 3,000 points.

[5] To prevent electromigration effects from taking place, wire width (for wires both within the block and between blocks) is also a parameter here, which self-adapts to the current through the wire, according to the layer's maximum current density specified by the technology.

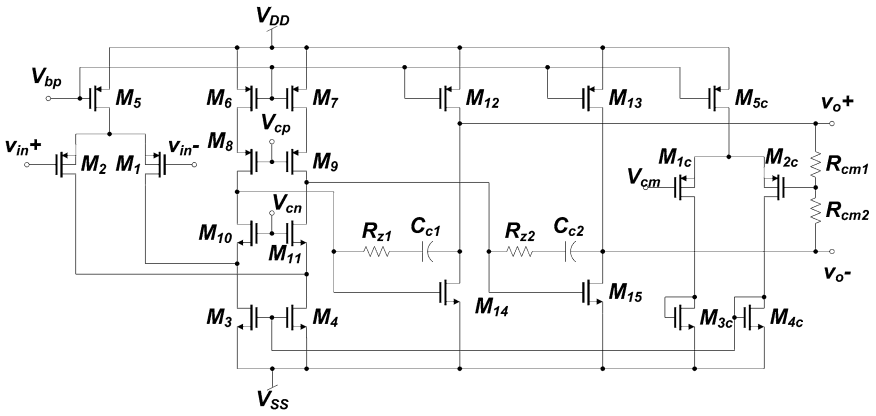


Fig. 6.3 Opamp used to illustrate the concept of slicing floorplan ([30], ©IEEE 2008)

In the second phase (top-down shape propagation), all the points (i.e., rows in the floorplan-sizing matrix) that do not meet the geometric goals defined by the user are removed. The following geometric goals are considered:

1. Aspect ratio, AR$=W/H$, with an acceptable deviation Δ_{AR}, such that the final layout aspect ratio remains bounded (AR$-\Delta_{\mathrm{AR}} \leq W/H \leq \mathrm{AR}+\Delta_{\mathrm{AR}}$).
2. Maximum and/or minimum width and/or height values.

For instance, if the user-specified geometric goal is to attain an aspect ratio between 0.9 and 1.1, the solution with minimal area can be found from the shape function in Fig. 6.4a and the zoomed-in view in Fig. 6.4b, which also shows the area occupation for these solutions. The optimum solution corresponds to the point in the shape function with lower area occupation, as indicated in Fig. 6.4b. In this way, the row with minimal area is selected from the floorplan-sizing matrix. Next, the slicing tree is traversed top-down to obtain the corresponding shape (width and height) of each leaf node. From these, the values of their GPs[6] can be retrieved.

The description provided above is to show how the GC works for every circuit that the optimization engine proposes to be evaluated by the simulator. Once a new circuit is proposed by the optimization engine, and before simulation, the GC module analyzes the sizing and returns the values of the geometric parameters that ensure minimal layout area (that is, the evaluated design has minimal area when considering all layout realizations coming from the different values that each block's GPs can take), minimal area loss, and the specified geometric goals.

Area occupation is minimized at the template level by the GC module. But, since a precise calculation of the area of the template instance is available right after the application of the GC module at each iteration, area is also minimized by

[6] Note that other layout features such as device symmetries are not targeted by the GC module; rather, these other layout aspects are embedded in the layout template itself.

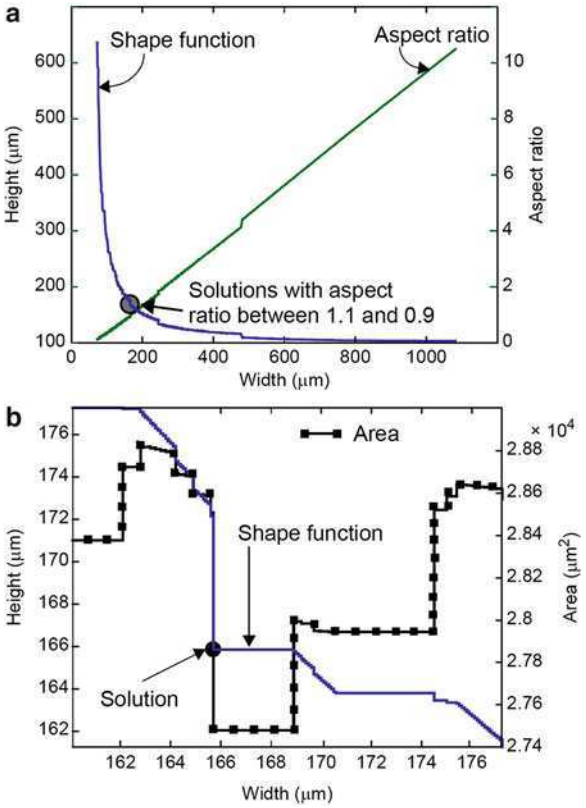


Fig. 6.4 (**a**) Example of global shape function for the analog cell in Fig. 6.3, and (**b**) area values for each point of its shape function ([30], ©IEEE 2008)

including it in the formulation of the cost function, as defined in (6.3). In this way, the evolution of the optimization algorithm will tend to minimize area, pretty much in the same way that other design objectives, e.g., power consumption, are minimized. The inclusion of area loss as an optional, complementary design objective serves the purpose of getting layout realizations that are very compact. Area loss can be also included as a general design objective in the electrical optimization-based sizing process.

6.4.3 Experimental Results

Four different experiments, with different geometric goals but the same set of electrical performance specifications and design objectives, have been carried out to validate the proposed solution. The demonstration circuit used is the opamp of

Fig. 6.3. The geometric parameters considered are the number of folds of transistors and resistors, and the side length of the unit rectangular capacitor implementing C_{c1} and C_{c2}. From a close inspection of Table 6.2 and the schematic of the opamp in Figs. 6.2 and 6.3, the reader can notice that analog layout quality aspects, such as symmetries in the signal path, were taken into account in the placement of the layout template. The design variables for the set of experiments are the widths and lengths of transistors M_1, M_3, M_5, M_8, M_{10}, M_{12}, and M_{1c}, the nominal value and strip width of resistors R_{cm1} and R_{z1}, the nominal value of capacitor C_{c1}, and the value of the biasing current. The variation ranges of these optimization variables define the design space to be explored. Constraints are also added in the optimization process. These constraints are circuit-specific design knowledge that is used to guide the search in the complex design space. An example of such a constraint is the one imposed on the aspects of M_{3c} and M_{4c} for current sources M_3 and M_4 to provide enough current for common-mode stabilization. The first two columns in Table 6.3 show the electrical design specifications (constraints, as defined in (6.2)) for the four experiments. Table 6.4 shows the geometric constraints. In all four experiments, the amplifier drives a resistive load of 50 kΩ and a capacitive load of 5pF. The design objectives that are defined for the set of experiments are the minimization of area, the minimization of power consumption, and the minimization of area loss. The simulator used (see Fig. 6.1) is HSPICE™. The optimization results are shown in Tables 6.3–6.5. In all four experiments, all electrical performance specifications are met, as shown in Table 6.3. Table 6.4 also shows the obtained values for the geometric goals of the resulting opamp layouts, and Table 6.5 lists the attained design objectives. All geometric goals have been successfully addressed, whereas minimization of layout area and power consumption have been carried out.

Table 6.3 Specified and obtained electrical performances in the GA-only experiments

Specification	Goal	Exp. #1	Exp. #2	Exp. #3	Exp. #4
DC gain (dB)	≥85	110.37	89.85	105.65	100.82
Unity-gain frequency (MHz)	≥50	65.79	50.64	75.95	50.37
Phase margin (deg)	≥50	57.26	71.78	52.93	68.94
CMFB loop DC gain	≥85	114.0	92.39	108.79	111.04
CMFB loop UGF (MHz)	≥25	27.64	25.15	29.28	27.52
CMFB loop phase margin (deg)	≥50	50.13	50.06	50.46	52.25
Output swing (V)	≥5.5	5.86	5.85	5.87	5.76
Slew-rate (V/μs)	≥55	59.15	57.25	71.72	59.96

Table 6.4 Specified and obtained values of the geometric goals in the GA-only experiments

	Aspect ratio		Width (μm)		Height (μm)	
Experiment	Constraint	Obtained	Constraint	Obtained	Constraint	Obtained
Exp. #1	[0.91,1.1]	1.00	–	165.7	–	165.85
Exp. #2	[1.95,2.05]	1.96	–	229.0	–	116.7
Exp. #3	–	–	<150	143.2	–	171.1
Exp. #4	–	–	<300	186.9	<150	138.1

Table 6.5 Attained design objectives and achieved area loss in the GA-only experiments

Feature	Goal	Exp. #1	Exp. #2	Exp. #3	Exp. #4
Power (mW)	Minimize	3.18	3.21	4.93	2.98
Area (μm^2)	Minimize	27,481	26,724	24,502	25,811
Area loss (% of area)	Minimize	1.94	3.06	0.93	0.57

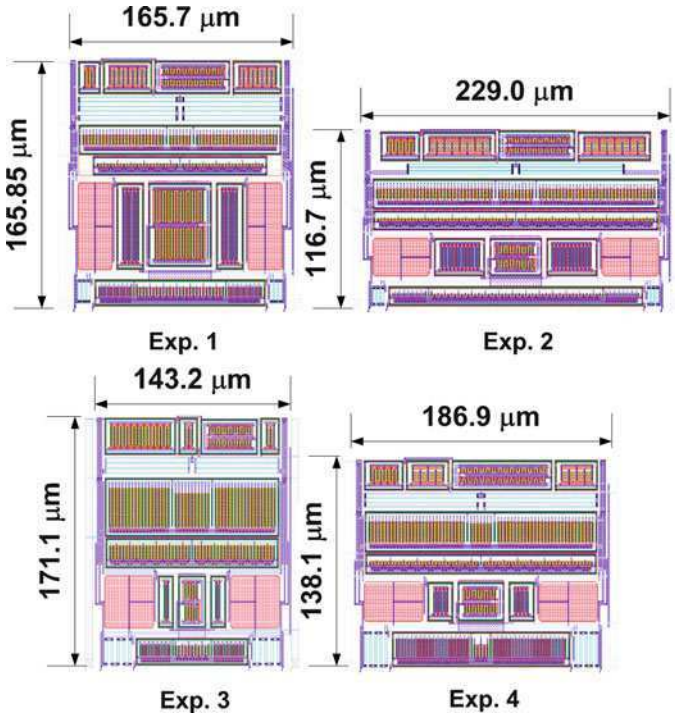


Fig. 6.5 Resulting layouts from geometrically constrained sizing experiments ([30], ©IEEE 2008)

Table 6.6 Number of iterations and elapsed CPU time in the GA-only experiments

Figure	Exp. #1	Exp. #2	Exp. #3	Exp. #4
Number of iterations	1,708	5,778	6,462	4,824
CPU time (s)	409.92	1380.9	1563.8	1157.76

The physical implementation for each experiment is shown in Fig. 6.5. Finally, Table 6.6 displays the elapsed CPU time[7] and the number of iterations of each experiment.

[7] The experiments were performed on a Pentium IV at 1.3 GHz.

6.5 Layout-Aware Sizing of AMS Circuits

6.5.1 Completing the Layout-Aware Sizing Methodology with Parasitic Extraction

One important fact in analog design is that accurate estimates of layout-induced parasitics can be only obtained with all the information on the eventually implement circuit layout, with complete information on the values of each and every one of the geometric parameters. The GC module in Fig. 6.1 determines the value of geometric parameters prior to parasitic extraction. Such information is then processed to estimate layout-induced parasitics. Afterward, the electrical description of the circuit is completed with the parasitic estimates and the overall performance is checked against intended performance specifications, which requires evaluation of the circuit performance.

Another important fact about parasitics is that many different extraction techniques exist [24]. The tradeoffs that these techniques expose relate accuracy and computation time. For MOS transistors, extraction can be done by using geometric methods and analytical methods. Geometric methods directly measure diffusion areas and perimeters. Analytical methods require having drain and source areas and perimeters as functions of the number of fingers, the exact implementation style, and fabrication process data. For interconnects,[8] parasitics can be extracted by using numerical, A–G (analytical–geometrical), or table lookup methods. Numerical methods try to solve the Laplace equation over the system of stratified layers. A–G methods use analytical parameterized models[9] for a number of commonly encountered interconnect configurations. Geometric methods are then used to obtain the layout parameters directly from the layout geometries to evaluate each analytical function and, thus, extract the capacitive parasitics. Lookup tables store the data generated by numerical simulations or experimental measurements.

Regarding the accuracy-time tradeoff, numerical methods are slower (due to the high computational resources demanded), but are accurate. Table lookup methods can be reasonably accurate estimates with relatively low computation times, but the data storage requirements grow very rapidly with the number and range of parameters describing a given interconnect configuration (that is, the higher the accuracy required, the larger the data set required). On the other hand, A–G methods are relatively fast and accurate.

Two approaches for parasitic estimation have been implemented in this layout-aware approach. In the first one (approach A), parasitic extraction is done by using geometric methods for transistors and 3-D A–G methods for interconnects and other

[8] Resistive and capacitive devices, such as poly-silicon or well resistors and PIP or MIM capacitors, can be treated as interconnect layers when extracting their related parasitics.

[9] The complexity of these models (and, thus, the accuracy of the parasitic extraction) usually refers to the dimensionality of the spatial configuration of stratified layers, such as the 2-D model, 2.5-D model, or the more complex 3-D model.

devices, which are pretty much the same methods used by most commercial parasitic extractors. The main objections with these extraction techniques is that, although the parasitics obtained are very accurate, these techniques have usually been considered to be slow to be included within an optimization loop [24, 27], and that they may yield long simulation times due to the presence of the large number of parasitics that are extracted.

The second approach (approach B) to parasitic estimation relies on analytical methods for the calculation of the MOS transistor diffusion areas and perimeters. For every layout style implementation for single, stacked, or interdigitized transistors, a set of equations were developed following the approach in [38]. These equations relate the diffusion area and perimeter of a transistor to its width, length, number of fingers, layout style, and fabrication process. With the equations, it is then possible to accurately compute diffusion capacitances for every transistor size. Estimates for the routing parasitics in this second approach are obtained by following a layout template sampling technique. Layout sampling has been reported in the literature with different approaches and purposes [5, 27, 28]. In our approach, a number of different instances of the circuit layout template are generated prior to any circuit sizing. This generation is done by sampling each of the n layout parameters (i.e., circuit design variables and geometric parameters) with m data points. The interconnect parasitics of each instance are extracted with a 3-D A–G extraction technique (with a commercial parasitic extractor), and stored in a lookup table. This table, relating the sizing and geometric parameters to the values of the routing parasitics,[10] can then be used in the sizing process to retrieve the values of these parasitics. The main issue of this approach is that, while it allows a very fast evaluation of critical parasitics, the number of design variables, being relatively large, makes it very time-consuming to generate and store the m^n possible instances and, also, the for the lookup table may be exceedingly large. To ease this issue, two sampling steps are applied. In the first one, a reduced number of instances (e.g., 100) is generated. Electrical simulation of these instances allows identifying and eliminating noncritical parasitics (those with negligible impact on the electrical performances of importance). For instance, the application of this technique to the fully differential operational amplifier in Fig. 6.6 provides the relevant parasitics shown in gray in the same figure.

The second step performs a denser sampling of only the relevant layout parameters, i.e., those parameters that are associated with the parasitics with a more significant impact on performance (this knowledge is obtained from the layout template). For those layout parameters with a nonsignificant impact on the critical parasitics identified in the previous step, a single sample is used. As a result of these two steps, only N instances (with $N \ll m^n$) are generated and extracted, and for those N instances, only the critical parasitics are stored.

[10] Note that the generation of the lookup table must be performed only once for the circuit's template; the table remains valid for any sizing process applied to that circuit.

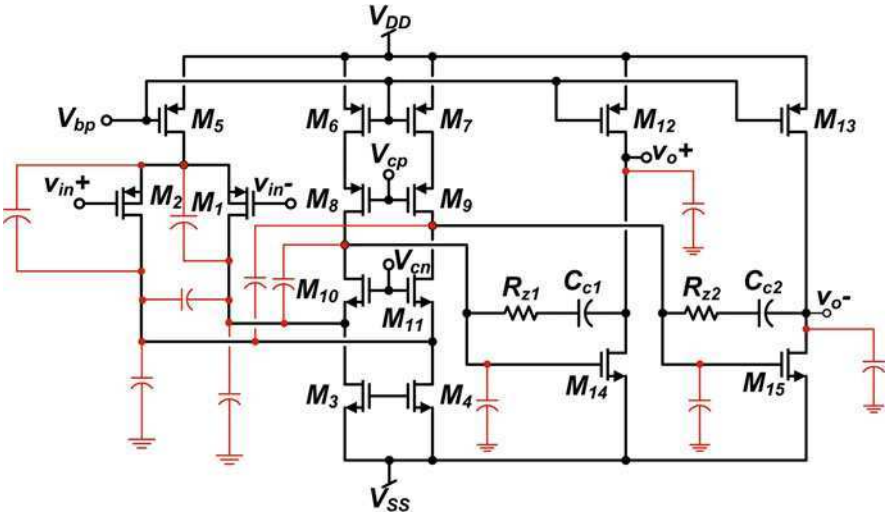


Fig. 6.6 Case study for the validation of the layout-aware sizing technique ([30], ©IEEE 2008)

Table 6.7 Specified and obtained electrical performances in the parasitic-aware experiments

Specification	Goal	Exp. I	Exp. II	Exp. III
DC gain (dB)	≥110	110.0 (111.0)	113.0 (113.0)	111.7
Unity-gain frequency (MHz)	≥90	91.8 (89.8)	105.7 (105.4)	106.6
Phase margin (deg)	≥65	67.6 (63.4)	65.4 (65.0)	66.2
Output swing (V)	≥5.25	5.3 (5.3)	5.3 (5.3)	5.4
Slew-rate (V/μs)	≥40	46.9 (46.8)	57.2 (57.0)	56.7
Power (mW)	Minimize	7.3 (7.3)	8.0 (8.0)	8.3
Area (μm × μm)	Minimize	195.8 × 358.8	173.8 × 191.25	189.6 × 193.05
Aspect ratio	≈ 1	0.55	0.91	0.98

6.5.2 *Experimental Results of Layout-Aware Sizing*

To illustrate the layout-aware sizing technique proposed, three sizing experiments have been carried on the operational amplifier shown in Fig. 6.6, with performance specifications listed in Table 6.7. These experiments are:

- *Experiment I*. In this experiment, no geometry or parasitic-related information is used. Global area minimization is pursued by minimizing the sum of the sizes of all devices (e.g., the width-length product of a transistor). Since no geometry-related aspect is used, no geometry goals (such as layout aspect ratio) can be enforced.
- *Experiment II*. This experiment considers both geometry and parasitics, with parasitic extraction following approach *B* explained above.
- *Experiment III*. This experiment considers both geometry and parasitics, with parasitic extraction following approach *A* explained above.

In all three experiments, the amplifier drives a 100-kΩ resistive load and an 8–pF capacitive load. Table 6.7 shows the results of each experiment. For experiments I and II, the performance characteristics obtained from electrical simulation of the extracted layout using a commercial extractor are shown between parentheses. For experiment III, the performances match since the extraction methods are the same. Note that, although nominal performances (without including the impact of layout parasitics) are all within specifications as shown in Table 6.7, some violations of specifications may arise when actually including parasitics (see the nonfulfilled phase margin and unity-gain frequency of experiment I). In these cases, additional redesign iterations are certainly required.

The final layout instances of the three experiments are shown in Fig. 6.7. Remarkably, when no geometry information is included, the layout implementations may end up with large empty areas and poor compaction, as it can be seen in the layout solution of Experiment I (GPs in this experiment were set to their default values).

Finally, Table 6.8 shows the number of iterations and the CPU times in the three experiments. Note that the use of approach *A* for parasitic extraction (Experiment III) requires approximately 15% more CPU time than when using the *B* approach (layout instancing and extraction takes 17% of the total sizing time). The benefit is that despite this increase in the CPU time all performance characteristics are

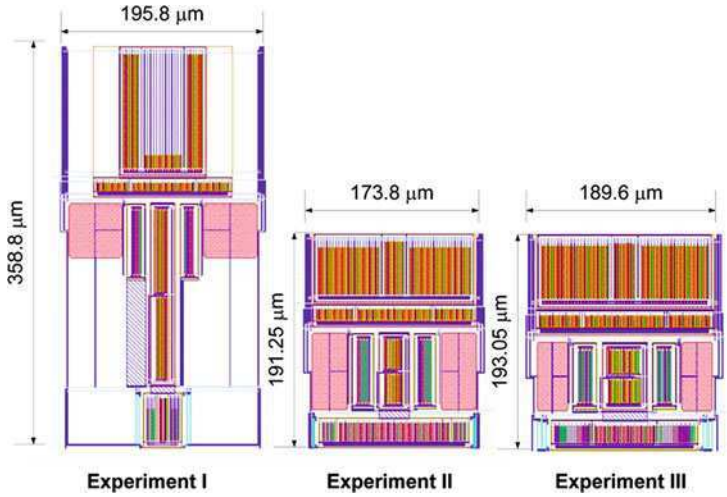


Fig. 6.7 Layout instances of experiments in Table VII ([30], ©IEEE 2008)

Table 6.8 Number of iterations and elapsed CPU time for the parasitic-aware experiments

Figure	Exp. I	Exp. II	Exp. III
Number of iterations	2116	2437	3044
CPU time (s)	507.8	612.31	880.3

guaranteed to be within specifications. Therefore, these results demonstrate that, at least for analog cells with a few tens of devices, the first approach for parasitic extraction is also reasonably fast to be included within an iterative optimization loop.

6.6 Conclusions

A layout-aware sizing methodology for analog circuits has been described in this chapter. This methodology minimizes the iterations between electrical and physical design phases in traditional design methodologies.

This is a flexible solution because it uses simulation-based optimization approach, which can be applied to many different types of analog circuits. Moreover, accurate evaluation of the circuit performance characteristics is guaranteed because of the electrical simulator used in the optimization loop. The inclusion of parasitics in the electrical sizing process ensures that the design solutions that are attained will meet the required specifications. Also, area is realistically minimized, both at the template level and at the global level, during circuit sizing because all layout-related geometrical information is included in the optimization. Thanks to the use of a floorplan-sizing algorithm, geometric goals (such as a certain layout aspect ratio) can be used as well as design objectives in the optimization process.

Future work on this topic includes the extension of the layout-aware sizing methodology to larger circuits by introducing hierarchical decomposition and specification transmission. Also, the use of multiobjective optimization instead of single-objective optimization (used in this chapter) can provide a way to relate the trade-offs between electrical performance and the use and efficiency of the layout template for the complete design space, which can help introducing layout template selection depending on the region of the design space the exploration is taking place.

References

1. Int. Technology Roadmap for Semiconductors. [Online]. Available: http://public.itrs.net, 2005.
2. MEDEA+ Design Automation Roadmap. [Online]. Available: http://www.medeaplus.org, 2007.
3. M. Degrauwe, O. Nys, and E. Dijkstra, "IDAC: An interactive design tool for analog CMOS circuits," *IEEE J. Solid-State Circuits*, vol. SSC-22, no. 6, pp. 1106 – 1116, Dec 1987.
4. J. Conway and G. Schrooten, "An automatic layout generator for analog circuits," in *European Design Automation Conference*, Mar 1992, pp. 513 – 519.
5. R. Castro-Lopez, F. Fernandez, and F. Medeiro, "Generation of technology-independent retargetable analog blocks," *Analog Integrated Circuits and Signal Processing*, vol. 3, no. 2, pp. 157 – 170, Nov 2002.
6. N. Jangkrajarng, S. Bhattacharya, and R. Hartono, "IPRAIL – intellectual property reuse-based analog IC layout automation," *Integration, VLSI J.*, Nov 2003.

7. S. Bhattacharya, N. Jangkrajarng, and C. Shi, "Multilevel symmetry-constraint generation for retargeting large analog layouts," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 25, no. 6, pp. 945 – 960, Jun 2006.
8. S. Bhattacharya, N. Jangkrajarng, and C. Shi, "Template-driven parasitic-aware optimization of analog integrated circuit layouts," in *ACM/IEEE Design Automation Conference (DAC)*, Jun 2005, pp. 644 – 647.
9. N. Jangkrajarng, L. Zhang, S. Bhattacharya, N. Kohagen, and C. Shi, "Template-based parasitic-aware optimization and retargeting of analog and RF integrated circuit layouts," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2006, pp. 342 – 348.
10. J. Harvey, M. Elmasry, and B. Leung, "STAIC: An interactive framework for synthesizing CMOS and BiCMOS analog circuits," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 11, no. 1, pp. 1402 – 1417, Nov 1992.
11. G. Gielen, H. Walscharts, and W. Sansen, "Analog circuit design optimization based on symbolic simulation and simulated annealing," *IEEE J. Solid-State Circuits*, vol. 25, no. 3, pp. 707 – 713, Jun 1990.
12. P. Maulik, L. Carley, and D. Allstot, "Sizing of cell-level analog circuits using constrained optimization techniques," *IEEE J. Solid-State Circuits*, vol. 28, no. 3, pp. 233 – 241, Mar 1993.
13. M. Hershenson, S. Boyd, and T. Lee, "Optimal design of a CMOS op-amp via geometric programming," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 20, no. 1, pp. 1–21, Jan 2001.
14. F. Fernández, A. Rodríguez, and J. L. Huertas, *Symbolic Analysis Techniques: Applications to Analog Design Automation*. IEEE Press, New York, 1997.
15. C. Shi and X.-D. Tan;, "Canonical symbolic analysis of large analog circuits with determinant decision diagrams," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 19, no. 1, pp. 1 – 18, Jan 2000.
16. P. Vancorenland, G. V. der Plas, M. Steyaert, G. Gielen, and W. Sansen, "A layout-aware synthesis methodology for RF circuits," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2001, pp. 358 – 362.
17. W. Nye, D. C. Riley, A. Sangiovanni-Vincentelli, and A. L. Tits, "DELIGHT.SPICE: An optimization-based system for the design of integrated circuits," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 7, no. 4, pp. 501 – 519, Apr 1988.
18. G. Stehr, M. Pronath, F. Schenkel, H. Graeb, and K. Antreich, "Initial sizing of analog integrated circuits by centering within topology-given implicit specifications," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov 2003, pp. 241 – 246.
19. R. Phelps, M. Krasnicki, R. Rutenbar, L. Carley, and J. Hellums, "Anaconda: simulation-based synthesis of analog circuits via stochastic pattern search," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 19, no. 6, pp. 703 – 717, Jun 2000.
20. F. Medeiro, A. Pérez-Verdú, and A. Rodríguez-Vázquez, *Top-Down Design of High-Performance Sigma-Delta Modulators*. Kluwer, Dordrecht, 1999.
21. E. Ochotta, R. Rutenbar, and L. Carley, "Synthesis of high-performance analog circuits in ASTRX/OBLX," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 15, no. 3, pp. 273 – 294, Mar 1996.
22. H. Chang, E. Liu, R. Neff, E. Felt, and E. Malavasi, *Top-Down, Constraint-Driven Design Methodology for Analog Integrated Circuits*. Kluwer, Dordrecht, 97.
23. J. Cohn, D. Garrod, R. Rutenbar, and L. Carley, "KOAN/ANAGRAM II: new tools for device-level analog placement and routing," *IEEE J. Solid-State Circuits*, vol. 26, no. 3, pp. 330 – 342, Mar 1991.
24. K. Lampaert, G. Gielen, and W. Sansen, *Analog Layout Generation for Performance and Manufacturability*. Kluwer, Dordrecht, 1999.
25. M. Dessouky and M. Louerat, "A layout approach for electrical and physical design integration of high-performance analog circuits," in *IEEE First International Symposium on Quality Electronic Design (ISQED)*, Mar 2000, pp. 291 – 298.
26. H. Onodera, H. Kanbara, and K. Tamaru, "Operational-amplifier compilation with performance optimization," *IEEE J. Solid-State Circuits*, vol. 25, no. 2, pp. 466 – 473, Apr 1990.

27. A. Agarwal, H. Sampath, V. Yelamanchili, and R. Vemuri, "Fast and accurate parasitic capacitance models for layout-aware synthesis of analog circuits," in *Design Automation Conference and Test in Europe Conference (DATE)*, Mar 2004, pp. 145 – 150.
28. M. Ranjan, W. Verhaegen, A. Agarwal, H. Sampath, R. Vemuri, and G. Gielen, "Fast, layout-inclusive analog circuit synthesis using pre-compiled parasitic-aware symbolic performance models," in *Design Automation Conference and Test in Europe Conference (DATE)*, vol. 1, Feb 2004, pp. 604 – 609.
29. A. Pradhan and R. Vemuri, "Efficient synthesis of a uniformly spread layout aware pareto surface for analog circuits," in *22nd International Conference on VLSI Design*, Dec 2009, pp. 131 – 136.
30. R. Castro-Lopez, O. Guerra, E. Roca, and F. Fernandez, "An integrated layout-synthesis approach for analog ics," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 27, no. 7, pp. 1179 – 1189, Jul 2008.
31. R. P. Brent, *Algorithms for Minimization Without Derivatives*. Prentice Hall, Englewood Cliffs, NJ, 2002.
32. G. Zhang, A. Dengi, R. Rohrer, R. Rutenbar, and L. Carley, "A synthesis flow toward fast parasitic closure for radio-frequency integrated circuits," in *ACM/IEEE Design Automation Conference (DAC)*, 2004, pp. 155 – 158.
33. *Virtuoso Parameterized Cell Reference*, 4th ed., Cadence Des. Syst. Inc., San Jose, CA, 2000.
34. *SKILL Language Reference*, 6th ed., Cadence Des. Syst. Inc., San Jose, CA, 2004.
35. R. Otten, "Automatic floorplan design," in *ACM/IEEE Design Automation Conference (DAC)*, 1982, pp. 261 – 267.
36. L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs." *Inf. Control*, vol. 57, no. 2/3, pp. 91 – 101, May/Jun 1983.
37. H. Koh, C. Sequin, and P. Gray, "OPASYN: a compiler for CMOS operational amplifiers," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 9, no. 2, pp. 113 – 125, Feb 1990.
38. R. Naiknaware and T. Fiez, "Automated hierarchical CMOS analog circuit stack generation with intramodule connectivity and matching considerations," *IEEE J. Solid-State Circuits*, vol. 34, no. 3, pp. 304 – 303, Mar 1999.

Chapter 7
Constraint-Driven Design Methodology: A Path to Analog Design Automation

Göran Jerke, Jens Lienig, and Jan B. Freuer

Abstract Physical design for analog ICs has not been automated to the same degree as digital IC design, but such automation can significantly improve the productivity of circuit engineers. Analog design remains difficult to formalize due to a large amount of expert knowledge involved, such as sophisticated constraints that are specified manually and satisfied through manual layout. We therefore propose a constraint-driven design methodology – a suite of algorithms and methodologies to capture key rules governing analog layouts and to produce layouts that satisfy these rules. In this chapter, we identify major challenges in analog physical design, and relate them to constraints. We introduce techniques for constraint representation and highlight the essential components of a constraint-driven design methodology. Finally, we explain how constraint-driven design impacts a typical analog design flow, layout algorithms, and the overall physical design methodology.

7.1 Introduction

While physical design automation of analog IC design has seen significant improvement in the past decade, it has not advanced at nearly the rate of its digital counterpart. This shortfall is primarily rooted in the analog IC design problem itself, which is very complex even for small problem sizes [7, 16, 23, 29].

The quality of a design result is generally determined by the degree to which compliance constraints have been met and predefined design objectives achieved. Due to the lack of uniform representation and interpretation of design constraints in the analog design flow context, most of the constraints in today's analog designs are still specified and considered manually by expert designers (expert knowledge). Furthermore, analog constraints are often used implicitly (i.e., based on a designer's experience) rather than being explicitly defined, which prevents their effective use

J. Lienig (✉)
Dresden University of Technology, IFTE, 01062 Dresden, Germany
e-mail: jens.lienig@ifte.de

H.E. Graeb (ed.), *Analog Layout Synthesis: A Survey of Topological Approaches*,
DOI 10.1007/978-1-4419-6932-3_7, © Springer Science+Business Media, LLC 2011

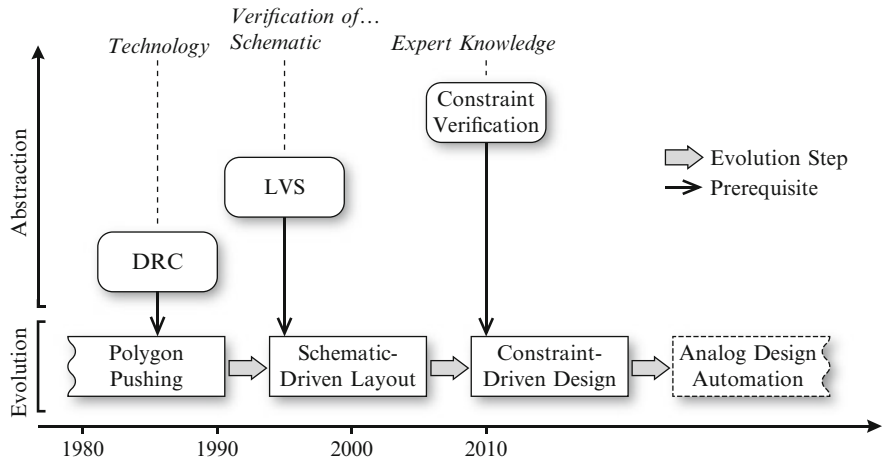


Fig. 7.1 The evolution of analog physical design methodologies, such as schematic-driven layout and constraint-driven design, towards the goal of a fully automated analog design flow

in design automation. However, progress in physical design automation for analog ICs is urgently needed as design sizes increase, along with significant challenges, such as increasingly stringent reliability and robustness requirements, and a rapidly widening verification gap.

Analog circuits are currently designed interactively, in terms of schematics, which are subsequently verified. Most researchers agree that this so-called schematic-driven layout (SDL) methodology will be replaced by analog design automation in the future, more in line with current practices for digital circuits. As we will show, constraint-driven IC design is both a necessary step toward full automation and also a precondition for it (Fig. 7.1).

The ultimate goal of fully automated analog design (analog design automation) can only be achieved if the schematic-driven design paradigm evolves into a constraint-driven design paradigm. This is based on the belief that we first need a methodology that allows for automatic inclusion of expert knowledge in the form of constraints, which also must be verified automatically. Only then one is able to tackle the task of analog layout synthesis in a comprehensive and consistent manner. In other words, the abilities of "analyzing" and "verifying" are a precondition for "synthesizing" [30].

This chapter provides an introduction to the concept of a constraint-driven physical design approach for arbitrary ICs in general, and for analog ICs in particular. First, we identify key similarities and differences between the physical design of analog and digital circuits, and the corresponding challenges, which we show are primarily constraint-related (Sect. 7.2). We discuss the constraint representation and classification in Sect. 7.3 and give an overview of the constraint-driven design flow and its essential components in Sect. 7.4. Here, we introduce fundamental components required in this flow, such as constraint representation, management, transformation, and verification. The application and resolution of

constraints, through constraint engineering, is discussed in Sect. 7.5. In Sect. 7.6, we then present the impact this methodology has on the overall IC design flow, the core design of design automation algorithms, and the required paradigm adjustments needed for analog physical design approaches. The chapter concludes with an anticipatory look at open problems (Sect. 7.7).

7.2 Problem Description

7.2.1 The Design Problem

In general, any (IC) design problem represents a complex and constrained optimization problem. The degrees of design freedom linked to the optimization problem span a multidimensional solution space, which is at least partially constrained by the given global design constraints. A feasible solution for a specific design problem is obtained by sequentially removing all degrees of design freedom while traversing and reducing the solution space and considering all context-relevant constraints and application profiles.

This reduction is done by sequentially transforming functional representations with many degrees of design freedom into equivalent ones with fewer degrees of design freedom. For example, using suitable methods one may transform a given functional specification into a netlist[1], which is subsequently transformed into a floorplan, a placement order, a wired layout and finally into a physical mask layout[2], which contains no further degree of design freedom.

Several functional transformations (design steps) can be active at the same time during analog IC design (Fig. 7.2). The strategy of how and when to remove a degree of design freedom during the design phase depends on several specific factors in

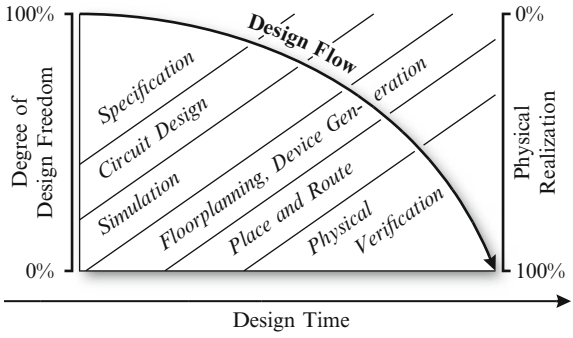


Fig. 7.2 Simplified design flow for analog IC design where design steps are typically overlapping. Multiple design steps are active at the same point in time [30]

[1] Functional representation of the given specification.

[2] Functional representation of the given netlist.

the design context. Among others, these factors may include the type of IC application, its usage profiles, reliability and robustness requirements as well as the current problem situation in a design phase with its linked constraints (design context).

In general, design constraints *must* be fulfilled, whereas design objectives *may* be fulfilled. A design objective that must be fulfilled hence represents a constraint, and must be treated as such. Similarly, any given design constraint that may be fulfilled should be considered as a design objective. The design goal is to achieve design results that fulfill all given constraints and which offer the highest level of achievement toward predefined design objectives.

7.2.2 Analog Vs. Digital Design Automation

Analog IC designs often contain only a small number of devices as compared to digital IC designs. Nevertheless, the effort required to design analog function modules often matches or even exceeds the effort for digital modules. This is mainly due to a much richer set of constraints that must be considered simultaneously (Sect. 7.3).

On average, each design object (instance, net, path, etc.) in an analog IC design must comply with a larger and more extensive set of constraints to fulfill its intended function (compared to digital design). The primary reason for this observation is the higher level of functional abstraction offered in digital designs. This allows digital designs to use fewer top-level constraints to guarantee a robust function.

Furthermore, the majority of constraints may yet be unknown when the analog design process begins. This renders automatic top-level design planning for analog IC designs nearly impossible. It is one of the reasons that highly skilled design engineers are still required to perform top-level design planning manually.

This constraint-related problem also makes algorithm and tool development for analog IC design much more difficult because the number of specific design algorithms may increase with each new type of constraint. Considering today's conventional design-algorithm development approach (one type of constraint and one algorithm to handle it), this approach falls short when it comes to linked constraints (Sect. 7.3). This represents one of the primary reasons why analog design automation is lagging behind its digital counterpart and why this gap is currently still growing.

Another important reason for the design gap is rooted in the level of completeness and consistency that can be applied to the consideration of constraints during IC design. Today's digital design tools already offer consistent and seamless design solutions. This is mainly due to their focus on a small set of various types of constraints, such as delay and clock skew. A unified description of constraints is not used in today's analog design tools and algorithms.[3] A common

[3] If not stated otherwise, the term "design algorithm" is subsequently used for both, design tools and their built-in algorithms due to their close relationship.

understanding of design implications due to constraints is not guaranteed with existing approaches. Hence, many analog constraints must still be considered manually or semi-automatically leading to their often inconsistent and noncomprehensive consideration.

Any inconsistent or noncomprehensive consideration of constraints widens the existing constraint verification gap. This gap exists because the design rule check (DRC) and the layout versus schematic check (LVS) do not include the verification of all constraints. A tremendous amount of research effort has already been expended for the tailored consideration and verification of special types of constraints, such as signal delay, device matching, and IR-drop. Nevertheless, a unified approach capable of dealing with all constraints during the entire design and verification phase is still missing.

Another difference between analog and digital IC designs is found in the way the functional transformations, i.e., the design steps, are linked and carried out. While most steps in digital IC design are separated from each other, the design steps of analog ICs are typically overlapping, and hence, tightly linked due to the impact of analog constraints (Fig. 7.2). For example, device generation, preplacement, and global routing usually occur simultaneously during the floorplanning phase of analog ICs. Analog design algorithms must thus consider various types of constraints simultaneously. This greatly reduces the impact of specialized design algorithms that handle only a small set of types of constraints.

To address the current shortcomings discussed in this section, a constraint-driven design approach is required that considers constraints in a comprehensive and consistent manner. Its cornerstones will be introduced in Sects. 7.4–7.6.

7.3 Constraint Classification and Representation

Constraints for IC design (hereafter, constraints) are classified by their complexity, category, form, and type. The classification criteria are discussed in this section.

From a formal point of view, constraints define relations between values of design variables (hereafter, variables). A relation between independent variables represents a simple constraint. Relations between dependent variables are denoted as complex constraints (Fig. 7.3). Constraints for IC design are linked to design objects, which represent data objects in the database of a design tool, such as cell, cellview, instance, net, terminal.

In general, constraints belong to one of the following four categories:

- *Technology* constraints enable manufacturing for a specific technology node (e.g., wire width, spacing, layer thickness).
- *Functional (electrical)* constraints ensure the intended IC functionality (e.g., maximum IR-drop between two net terminals, minimum gain, maximum offset voltage).
- *Design methodology (geometry)* constraints reduce the overall complexity of the design process. They also guide transformations, enforce a specific design

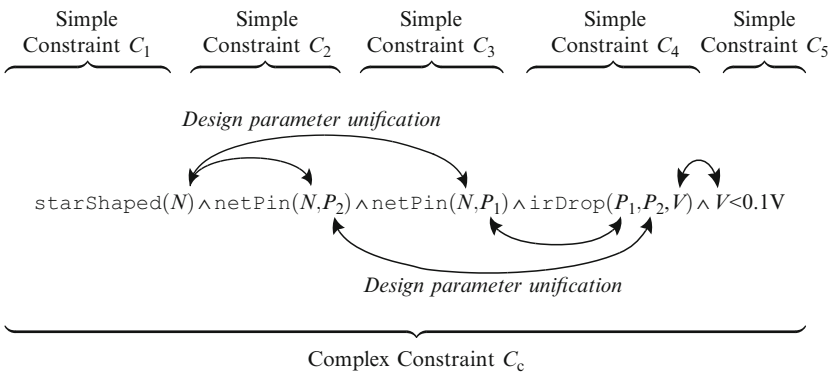


Fig. 7.3 Four simple constraints (`starShaped`(N), `netPin`(N, P_2), `netPin`(N, P_1), `irDrop`(P_1, P_2, V), and $V < 0.1$V) form a complex constraint C_c through a conjunction defined in constraint logic programming (CLP) notation. The complex constraint C_c is satisfied if all four simple constraints are satisfied. The simple constraints are tightly coupled through the design parameters N, P_1, P_2, and V that must be substituted (unified) to resolve C_c

pattern, or describe a context to which other constraints are associated with (e.g., maximum design hierarchy depth, maximum number of devices in a cluster, predefined layer for power-routing, bus width).

- *Commercial* constraints (e.g., maximum die area, number of layers).

A constraint is given in either an implicit or an explicit form. An implicit constraint is not clearly expressed and may be given as plain textual note or may arise from assumptions intrinsically built into circuit descriptions or design algorithms. Implicit constraints represent nonformalized design knowledge. Contrary to implicit constraints, explicit constraints are clearly expressed and represent formalized design knowledge. Examples of implicitly defined constraints are the placement requirements of differential pair transistors – they must be placed symmetrically to maximize device matching. While this is obvious to any layout designer, the inclusion of such complex rules into both layout and verification tools is often not possible for applications that contain additional requirements, such as parasitic interconnect matching. Hence, due to its nonformal nature, implicit constraints cannot be utilized for any type of controlled and automated constraint-driven design. On the other hand, explicitly defined constraints are accessible to design algorithms and thus are a primary requirement for any constraint-driven design flow.

Each constraint belongs to a specific constraint type that represents a classification property for the same class of constraints. The type of a constraint always corresponds to the type of the corresponding design variables. Constraint types have a clearly defined physical, electrical, mechanical, mathematical, or geometrical unit (e.g., the constraint type “IR-drop” has the unit Volt, the type “signal delay” the unit Seconds). The relevance and impact of a constraint type strongly depend on the specific design context.

To formalize design constraints, all constraints and all related design variables must be uniformly represented in an abstract form. The conversion of constraints into a uniform representation must be complete and unambiguous. A uniform representation enforces a common understanding of constraints among all involved design algorithms. Hence, it is a primary requirement for addressing the analog (constraint) design problem [11, 26]. Constraint logic programming (CLP) [8, 19] embodies a feasible approach for uniform constraint representations. In CLP, constraints are defined in the body of conditions (clauses) (Fig. 7.3). All constraint examples discussed in this chapter are based on the CLP notation.

Assume an IR-drop constraint $V_{\mathrm{IR}}(P_1, P_2) < 0.1\,\mathrm{V}$ stating that the IR-drop between two layout pins P_1 and P_2 must be less than 0.1 V. This functional constraint is simple since it is completely independent from any other constraint. If this example is transferred to a more formal representation, such as CLP, the IR-drop constraint must be written as a relation between design parameters. A possible representation is the relation $\texttt{irDropLessThan}(P_1, P_2, 0.1)$. However, this approach is very restrictive. For example, neither equality nor any other inequality can be expressed. To obtain a more general representation, it is advisable to split this constraint into a conjunction of a functional and an arithmetic constraint $\texttt{irDrop}(P_1, P_2, V) \wedge V < 0.1$ with V representing the actual IR-drop between pins P_1 and P_2.

The IR-drop between two net pins P_1 and P_2 is usually considered within a specific design context, in our case the net N, which owns both pins. This introduces two structural constraints $\texttt{netPin}(N, P_1)$ and $\texttt{netPin}(N, P_2)$. In addition, if the IR-drop needs to be considered only for nets with, for instance, a star-shaped layout topology, another structural constraint $\texttt{starShaped}(N)$ must be added. Figure 7.3 depicts the conjunction of these constraints that form the complex constraint C_{c}. The coupling of the simple constraints is obtained via substitution (unification) of the design variables N, P_1, P_2, and V (Sect. 7.5.1).

7.4 Components of a Constraint-Driven Design Flow

A design flow that considers all relevant constraints in a consistent and comprehensive manner is subsequently denoted as constraint-driven design flow. This flow requires several complementary design flow components that are shown in Fig. 7.4.

Constraint management provides the management of constraint data and the assignment of constraints to design objects (Sect. 7.4.1). To obtain design results meeting their specification, constraints are derived from design objectives (*constraint derivation*, Sect. 7.4.2). Constraints are transformed between the physical, electrical, or geometrical domain to be suitable for design algorithms in a particular design context (*constraint transformation*, Sect. 7.4.3). The *constraint sensitivity analysis (CSA)* determines the sensitivity of a design parameter with respect to related constraints. The CSA finds the most constraint-sensitive design parameters in a particular design context. Constraint sensitivity information can then be used

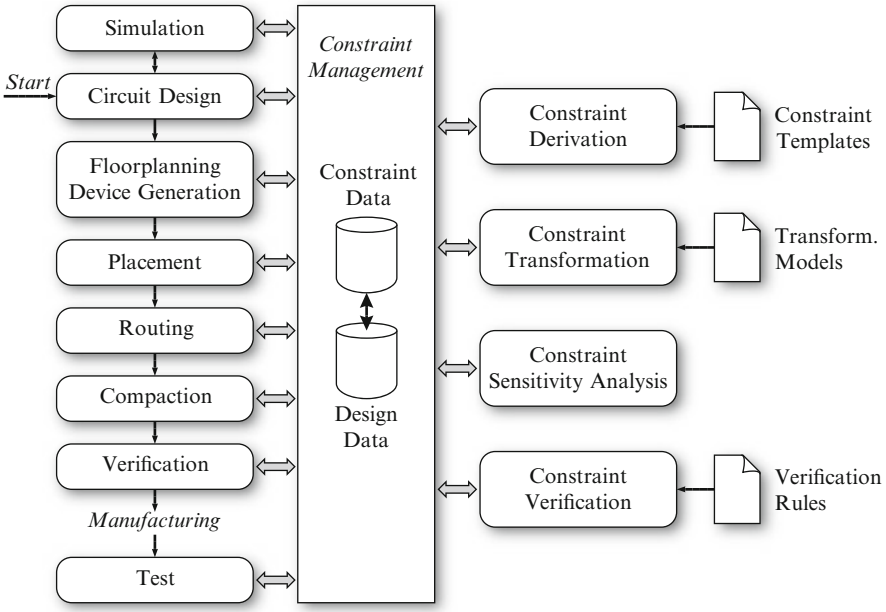


Fig. 7.4 Essential components of a constraint-driven design flow

to guide the design generation (Sect. 7.4.4). Finally, despite the use of a constraint-driven layout generation, the compliance of a design result with its given constraints must be verified using *constraint verification* (Sect. 7.4.5).

7.4.1 Constraint Management

The task of constraint management is to administer the storage of constraint data while synchronizing the link between constraints and design objects. The management system must also guarantee the semantic integrity of the constraints across different levels of abstraction, and support hierarchical relations between design objects and dependencies [6, 22]. In addition, it is responsible for keeping constraints consistent and valid, which requires close interaction with design databases as well as with constraint and design data manipulating design algorithms. Furthermore, constraint-driven design algorithms require fast access to constraint information through (standardized) application programming interfaces.

The detection of over-constraints is an important subcomponent of a constraint management system. It is made available by the constraint verification (Sect. 7.4.5). Over-constraints represent a condition in which not all given constraints can be fulfilled simultaneously. The related formal mathematical problem is denoted as constraint satisfaction problem (CSP). Over-constraints must be resolved by

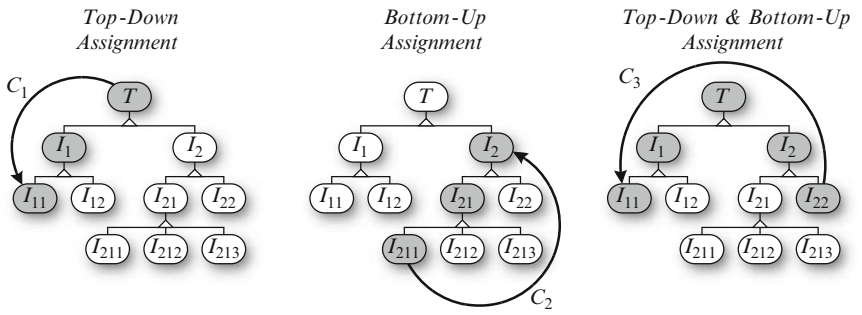


Fig. 7.5 Assignment of constraints to design objects in a design hierarchy tree. In this example, T represents a top cell incorporating several cellview instances I_1–I_{213}

constraint satisfaction methods, such as constraint propagation, constraint relaxation or backtracking, to obtain feasible design results [20]. The use of constraint weights as a decision criterion to resolve over-constraint conflicts is a common approach. However, this method is likely to become unsuitable if the number of constraints increases since many constraints may have equal or similar weights, thus making them unusable as decision criteria.

Constraint management also incorporates the assignment of constraints to design objects (a) in the existing design hierarchy, (b) across the extent of design objects and design steps, as well as (c) within a design hierarchy that is defined by design objects and linked constraints (virtual design hierarchy). The use of these assignment options strongly depends on the specific constraint. Furthermore, constraint assignment is either permanent or temporary depending on the particular design context.

The assignment of constraints within a hierarchical design can be either performed top-down, bottom-up, or combined top-down-bottom-up (Fig. 7.5). For instance, a net shielding constraint may be assigned from the I/O pad in the top cell down to a specific instance terminal in a subcell (top-down assignment). The shielding constraint is then assigned to all connected net objects in the design hierarchy.

The assignment can also be performed across the extent of design objects, such as instances. Using the previous example, cellview instances (e.g., metal resistors) must be skipped if the net shielding constraint is to be assigned to all nets that are physically connected on the chip mask. The net shielding constraint is then also hierarchically assigned to all subnets that would connect to the main net if the metal resistors were shorted.

In case the I/O pad is located in a subcell, then the shielding constraint must be assigned to all connected lower level nets as well as to all higher level nets that are connected to the I/O pad cell's instances (top-down and bottom-up assignment). Here, net shielding constraints are assigned within a virtual design hierarchy that is defined by the I/O pads' location in the design hierarchy tree and the hierarchical connectivity of the nets to be shielded.

During top-down assignment of a single constraint, only one constraint is assigned to each related design object in the cellviews that are traversed in the design

hierarchy tree. In contrast, the bottom-up assignment allocates as many constraints in the design hierarchy tree as instances of that cellview exist in the flattened design hierarchy, making it computationally more expensive.

7.4.2 Constraint Derivation

The process of deriving constraints from design objectives is denoted as constraint derivation or constraint generation. Design objectives are given as specification goals or requirements that must be met, but they can also arise from a local design context.

Design objectives are translated into constraints using (a) derivation rules, (b) deduction processes based on logic calculus, or (c) the designer's expert knowledge. The first two derivation methods can be applied with a high degree of automation in case the IC specification is given in a computer-processable form, such as an executable specification. The derivation process creates constraints belonging to the technology, functional, design methodology, or commercial constraint category (Sect. 7.3).

The rule-based derivation of constraints utilizes a fixed rule to transform a design objective into a set of constraints while considering the particular design context. The constraint transformation discussed in Sect. 7.4.3 is a form of indirect constraint derivation since it creates lower level constraints that depend on higher level constraints.

Deduction-based constraint derivation can be seen as a high-level extension of rule-based derivation methods. Here, a logic reasoning system draws conclusions from design and constraint data and then applies a set of constraint derivation rules to relevant design objects. For example, based on a logical conclusion that MOS and bipolar transistors both belong to the same category of devices "transistor", a specific constraint rule may be applied to both MOS and bipolar transistors, even in the case where the derivation rule was only defined for one transistor type. This functionality permits the development of high-level constraint derivation methods and offers an important level of abstraction required for the reuse of analog blocks.

Expert knowledge is still often required to translate critical design objectives into constraints. This is especially the case for global design objectives that would result in various sets of complex constraints and cannot be easily resolved by automatic rule-based approaches. Unfortunately, the expert knowledge only exists in an unstructured and nonformalized form. Nevertheless, making expert knowledge more accessible represents a good starting point for further analog design automation.

7.4.3 Constraint Transformation

Constraint transformation translates higher level constraints into a set of equivalent lower level constraints and vice versa (inverse constraint transformation) using

transformation rules [21]. Multiple transformation rules may apply for a specific higher level constraint resulting in different sets of lower level constraints. The choice of an appropriate transformation rule inherently constrains the solution space, thus reducing the number of global degrees of design freedom.

The choice of a transformation rule depends on the particular design problem and design context. Any transformation process must ensure a complete and unambiguous transformation result. The same applies to the inverse constraint transformation, which must be defined for constraint verification purposes (Sect. 7.4.5).

The transformation of constraints is based on a particular transformation model, which is translated into a set of transformation rules. Transformation rules for simple constraints are represented by independent equations. They contain the involved design variables in the higher transformation level and the variables in the lower level. Transformation rules for complex constraints are represented by a set of coupled equations containing all coupled design variables.

The relation of subconstraints specific to each complex constraint type is not affected by the transformation since the transformation of simple constraints only focuses on their specific context. This statement is made here since it is assumed that any transformation will only produce lower level constraints that do not affect higher level constraints. In the case where lower level constraints affect higher level constraints, design iterations are very likely to occur (i.e., the design steps must be reversed and redone with another design strategy).

In general, more than one transformation rule may exist for a particular type of constraint. The decision which transformation rule to use is specific to the design context, the design algorithm, and the applied design strategy. For example, suppose the functional specification of a circuit results in a specific maximum IR-drop between an I/O pad and a specific instance terminal in a subcell. Assuming that the current flow in the interconnect is known, the transformation of the IR-drop constraint may result in constraints for I/O pad and subcell placement and a corresponding set of routing constraints. A constraint-driven design algorithm can then decide whether the placement in this context is more critical to deal with than the routing and act accordingly (see also Sect. 7.4.4). For instance, in case the placement is fixed, the final transformation of the given IR-drop constraint would then yield a set of routing constraints and local degrees of design freedom (i.e., routing design parameters, such as wire length, layer, wire width). These can then be used by a routing algorithm to find a suitable interconnect layout.

7.4.4 Constraint Sensitivity Analysis

Constraint sensitivity analysis (CSA) determines the context-specific sensitivity of numerical design parameters with respect to related constraints. The CSA consists of two modules: a module that determines sensitivity of design parameters with respect to output parameters and a module that determines the relative distance of a design parameter value to its related constraints. Both modules provide valuable information that can be utilized by designers and by design algorithms.

The sensitivity analysis is based on a mathematical model, which describes the physical, electrical, or geometrical nature of a particular design subproblem. The model represents an equation system that contains all relevant design parameters and output parameters. Several approaches are reported to determine the sensitivity of design parameters. Among these approaches, local methods based on the partial derivatives of the model output parameter and statistical methods based on sampling, Bayesian and Monte Carlo methods are the most important ones [4, 18].

Considering a set of constraints $x_\mathrm{l} \leq x \leq x_\mathrm{u}$, the relative distance d of design parameter value x to a lower constraint boundary x_l and an upper constraint boundary x_u is determined as follows:

$$d_\mathrm{l}(x) = \exp(x_\mathrm{l} - x) - 1 \quad \text{and} \quad d_\mathrm{u}(x) = \exp(x - x_\mathrm{u}) - 1. \tag{7.1}$$

The parameter value x matches with the lower bound constraint value if the relative distance $d_\mathrm{l} = 0$. A constraint violation is detected in (7.1) if $d_\mathrm{l} > 0$ while no violation occurs if $d_\mathrm{l} < 0$. The same applies to d_u while considering the upper bound constraint value.

Design decisions can be made by design algorithms based on the sensitivity information of parameters, the relative distance of parameter values to related constraints and a given design strategy. Design algorithms may use that information in several ways. Depending on the design strategy, a design algorithm may point its focus to the fixation of design parameters with a high sensitivity toward an important output parameter or it may focus on low sensitivity parameters. The information about the parameter distance lets the design algorithm recognize the severity of constraint violations. For example, design parameters violating related constraints may then be considered with a higher priority.

It is also of interest for a design algorithm to know which design subproblems are independent from each other. A low sensitivity of design parameters toward a common output parameter means that they are weakly coupled with respect to that output parameter. The sensitivity analysis can be used as a method to identify local design task parallelism by searching for groups of design parameters and constraints that are either not or only weakly coupled. They can be dynamically partitioned into independent groups for which the next design step can then be performed independently from each other.

An example of a CSA application is given in Fig. 7.6. Here, a constraint sensitivity analysis is applied while routing a wire closely located to a heat source (e.g., a power transistor). Given an IR-drop constraint $V_\mathrm{IR} \leq V_\mathrm{IR-max}$, a design decision has to be made whether to move the wire away from the heat source, thus varying the interconnect temperature T, or to fix the wire width w. The design parameters and the constraint in Fig. 7.6 are denoted as follows: wire width w, length l, thickness d, reference temperature T_ref, IR-drop constraint $V_\mathrm{IR} \leq V_\mathrm{IR-max}$, $V_\mathrm{IR} = i \cdot \rho \frac{l}{w \cdot d} \cdot (1 + TK_1 \cdot (T - T_\mathrm{ref}))$, DC current i. A constraint violation is likely in case T is varied while $w \approx w_1$, whereas it becomes less likely in case $w > w_1$. To avoid an IR-drop constraint violation, the modification of the design parameter w is the primary choice if $w \approx w_1$ due to its high local sensitivity related to the

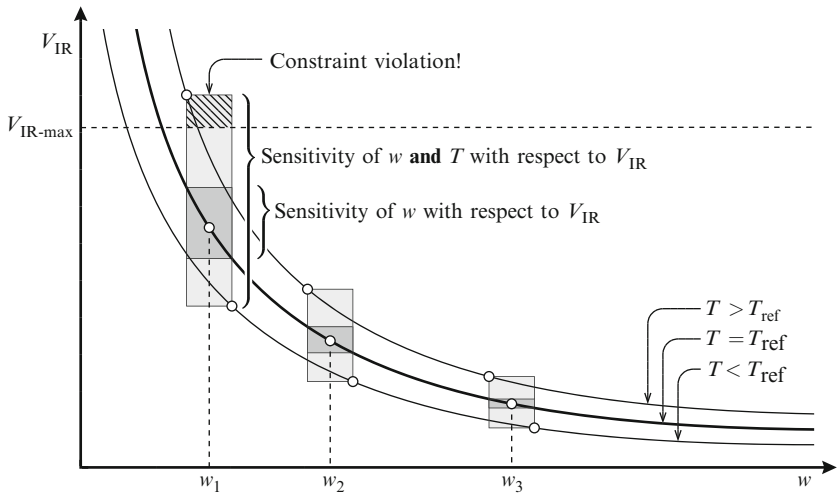


Fig. 7.6 A constraint sensitivity analysis is applied to parameters of a wire segment that is closely located to an on-chip heat source. A constraint violation is likely in case the interconnect temperature T is varied (by moving the wire's location) and a wire width w with $w \approx w_1$, whereas it becomes less likely in case $w > w_1$. To avoid an IR-drop constraint violation, the modification of w is the primary choice if $w \approx w_1$ due to its high sensitivity toward V_{IR} while w loses its sensitivity for $w \gg w_1$. (See text for parameter denominations and further explanation.)

output parameter V_{IR} while w loses its impact for $w \gg w_1$. If CSA is used as a filter to find all sensitive design parameters, then w is only required to be considered if $w \ll w_2$.

The CSA allows designers to study the impact of local design decisions and to trace root causes in case compliance requirements cannot be met by the given set of constraints. Sensitivity analysis is the key to the power of decision analysis in situations where the influence of design parameters is not known precisely, since it considers the design context in which constraints apply. As is obvious from this explanation, the availability and application of the CSA allows new approaches for algorithm development and analog design automation.

7.4.5 Constraint Verification

Constraint verification comprises the verification (a) whether a set of constraints is fulfilled for a particular design result and (b) whether a given set of constraints raises mutual conflicts (over-constraint, Fig. 7.7). Constraint verification represents a key component of the constraint-driven design flow. This is due to its formidable contribution to reduce the verification gap discussed in Sect. 7.2.2. Constraint verification ensures correct application functionality, and it is essential to improve design quality, reliability, and robustness. Commercially available constraint verification

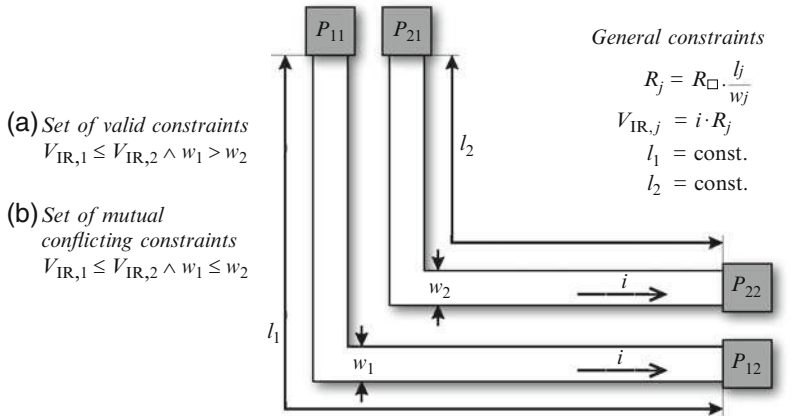


Fig. 7.7 Constraints illustrated in a two-net topology. A DC current i is present in both wires leading to a static IR-drop voltage V_{IR}. While the set of constraints in (a) is feasible, the two constraints in (b) are mutual conflicting (over-constraint). Here, a smaller IR-drop within one of two wires cannot be achieved if this wire is not allowed to be wider than the other one

tools with yet limited verification capabilities currently comprise Mentor Graphics Calibre® PERC [24] and the constraint verification engine integrated into Cadence Virtuoso® IC 6.1 [5].

As mentioned earlier, a rich set of constraints must be considered during the design of analog ICs. A significant fraction of these constraints are complex constraints, whose fulfillment cannot be verified with conventional verification approaches. This is due to the fact that all of today's verification approaches require one specific verification algorithm for each type of constraint. Clearly, conventional constraint one-to-one verification approaches (one verification algorithm for one type of constraint) are not feasible for the complete verification of analog IC designs. Making matters worse, many constraints (and constraint types) are still unknown at the beginning of the design process.

An approach to address the verification problem for complex constraints, the "meta-verification approach", was introduced in [11] and is discussed in more detail in Sect. 7.5.2. The core idea of meta-verification is that each complex verification problem can be subsequently resolved into smaller and usually independent verification subproblems. These subproblems can then be addressed using existing verification algorithms. The meta-verification references functionality accessible from external tools (e.g., design data access or specialized verification functions offered by a particular tool) to perform verification tasks. The meta-verification framework creates an abstraction layer around multiple design and verification tools, and it manages correct execution of the defined meta-verification tasks.

The CLP-based verification approach in [11] is capable to address independent as well as coupled, i.e., dependent verification problems. It also allows the detection of mutual constraint conflicts (over-constraints) by drawing logical conclusions from the given constraint and design data information. The approach is described in more detail in Sect. 7.5.

The definition of verification tasks for a meta-verification system to check constraint compliance is generally done as follows. First, the constraint verification task is defined and formalized. The formal description of a verification problem is then translated into a set of constraint verification rules. Finally, the verification rules are used by circuit and layout designers to perform constraint verification tasks. The application of these rules may depend on the design context of the particular constraint verification problem.

Significant effort must be taken by PDK developers and designers to develop, optimize, and verify the set of rules for constraint derivation, transformation, and verification. The sequence in which subverification tasks are processed has a significant impact on the required overall time for constraint verification. For example, suppose there are short-running and long-running subverification tasks defined in a specific CLP-based meta-verification rule. If feasible for a particular verification task, it is beneficial to shift all long-running subverification tasks to the end of that rule in order to execute them later than the short-running subverification tasks. Subverification tasks are not executed if a previous subverification task of a rule already revealed constraint violations. This approach will effectively prevent unnecessary and potentially long-running subverification tasks from being executed. As obvious, verification rule development and optimization requires a deep understanding of the underlying verification task.

Practical application of the meta-verification approach has revealed that the required initial effort is comparable to the effort needed for the development of DRC and LVS rule sets [11]. The reuse of rules for constraint derivation and meta-verification is simple and efficient since, in general, data and rule abstraction can be used for technology, design, and constraint data (Sect. 7.5).

Constraint verification is divided into static and dynamic constraint verification, based on the constancy of the constraint and design data. The corresponding constraint satisfaction problems (CSP) which are to be solved are denoted as static CSP and dynamic CSP [15]. For example, any sign-off verification of an IC design must be based on constant design and constraint data, hence, static constraint verification is applied in this case. Nevertheless, constraint-driven design algorithms can also use constraint verification for specific "what–if" analyses. Since these algorithms can change design and constraint data during their analyses and during the design step, the related constraint verification is based on dynamic data. Hence, the latter case represents dynamic constraint verification. Both, static and dynamic constraint verification can be applied either to the full set of constraint and design data, or to a design-context specific subset.

The required overhead for static constraint verification is typically significantly smaller compared to dynamic constraint verification. The additional overhead in the latter case is primarily caused by a cumulative data latency effect that occurs if design and constraint data are frequently accessed by design algorithms and/or the verification framework. Hence, low-latency access to design and constraint data will significantly speed up dynamic constraint verification. For static constraint verification, design and constraint databases are usually accessed only once during initialization, thus mostly avoiding data access latency issues.

7.5 Constraint Engineering

The application and handling of constraints during the IC design process is denoted as constraint engineering. In this section, we first provide a brief overview of computational approaches to address the constraint resolution problem (constraint programming). We then introduce the constraint engineering system (CES), which represents a framework that combines several constraint programming approaches in a single software framework. This framework integrates the previously discussed design flow components into a unified design environment, which facilitates the inter-operability between these components. It also increases the ability to perform design tasks on a higher level of abstraction, thus enabling new possibilities for analog design automation.

7.5.1 Constraint Programming

Constraint programming represents a programming paradigm where relations between (design) variables are stated in the form of constraints. These relations form a constraint satisfaction problem, which is resolved by constraint solvers.

The resolution of constraints usually occurs when multiple constraints are simplified or when the existence of one or more constraints leads to new (lower level) constraints. The constraint engineering uses specialized constraint solvers to handle all aspects of the constraint handling. The specialization is required since the handling strongly depends on the domain (or type) of the constraints. For example, a boolean constraint must be handled differently than a constraint that is defined over real numbers. The solving of arithmetic constraints, for instance, highly depends on the constraint complexity, such as linear or polynomial. Different constraint-solving approaches exist that are tailored to address various constraint satisfaction problems [3, 31].

As mentioned before, the formal constraint representation is a key requirement for a constraint-driven design flow. A formalism is required to describe the interaction of constraints, which are mainly the constraint derivation (Sect. 7.4.2) and constraint transformation (Sect. 7.4.3).

There are many approaches where constraints have been integrated into traditional programming languages [1, 10, 17, 27]. Due to the stateless character of constraints, the family of constraint logic programming (CLP) languages [8, 9, 32] is the natural choice for the formalization of constraints. The declarative logic calculus approach in CLP has also the advantage that only the problem has to be formalized but not its solution. Compared to other constraint programming approaches, the application of CLP significantly reduces the effort needed to provide the required rules for constraint derivation, transformation, and verification. Therefore, the core of the constraint engineering system discussed in the next section is based on a logic calculus engine (Fig. 7.8).

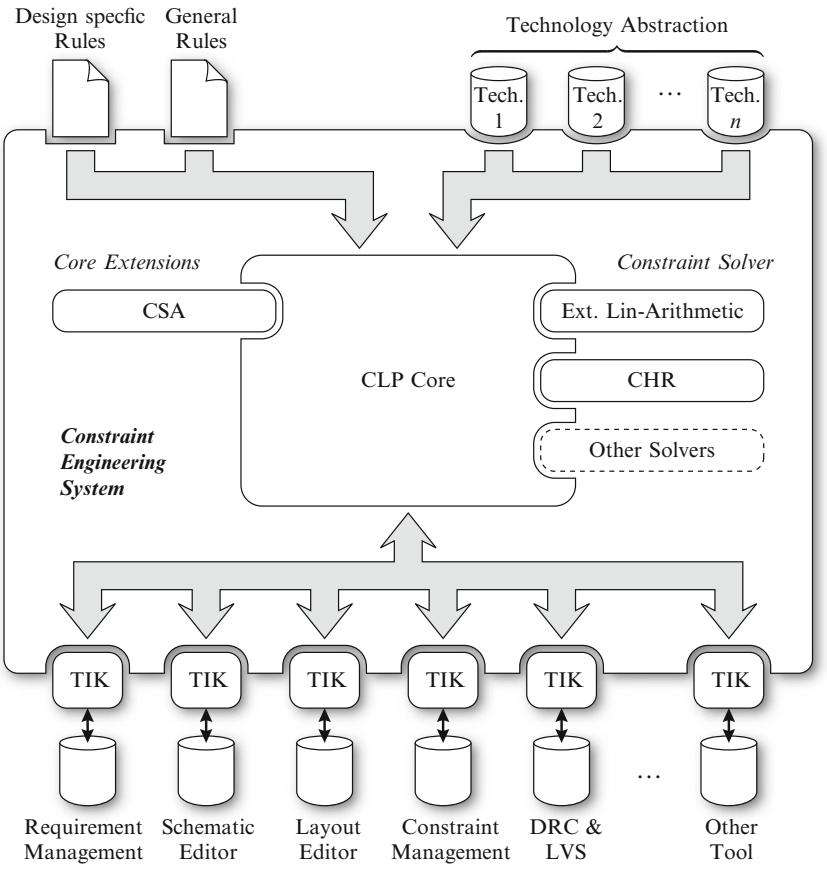


Fig. 7.8 Architecture and data flow of the constraint engineering system (CES). The tool integration kits (TIK) transform tool-specific data into the CLP language and vice versa

With the introduction of constraint handling rules (CHR) in 1991 [12], it became very easy to define new constraint solvers that are perfectly tailored to a specific constraint problem. Via CHR, new constraint solvers can be defined through two different kinds of handling rules. The propagation rule creates one or more constraints from a given set of constraints. Assume for instance the less-equal constraint "$\leq$". If there already exist two constraints $A \leq B$ and $B \leq C$, a suitable propagation rule would derive the constraint $A \leq C$. The second rule represents a simplification rule that removes one or more constraints from a given constraint set. Regarding the previous example, assume that there are two constraints $A \leq 5$ and $A \leq 7$. The simplification would remove $A \leq 7$ since it is overridden by $A \leq 5$. Due to efficiency reasons, there is usually also a third rule in CHR, the "simpagation". It combines simplification and propagation within a single rule.

7.5.2 The Constraint Engineering System

The application of constraint engineering is an important step toward a constraint-driven design flow. A flexible software architecture is required to integrate the new design flow components introduced in Sect. 7.4. Our approach of such an architecture will be subsequently denoted as constraint engineering system (CES), whose structure is depicted in Fig. 7.8 [11].

The CES is designed to act as a middleware between various design tools that offer an accessible application programming interface. The CES core engine is capable of making logical decisions based on multiple knowledge bases, which are provided from various external sources.

As shown in Fig. 7.8, the CES is based on a plug-in architecture that allows the flexible extension of its functionality. An extension point of the CES regards the access to all design tools that are accessible within the existing design flow. A translation layer, denoted as *Tool Integration Kit* (TIK), transforms the tool-specific data into logic calculus knowledge using CLP language so that it can be accessed by meta-verification. Vice versa, the TIK also provides the functionality of transferring data from the meta-layer back to the connected design tool. This allows the back-annotation of constraints that were processed in the CES to an external design tool. A TIK also enables a high-level access to the functionality of a design tool, which can then be utilized by particular design algorithms or the constraint verification. For example, a schematic entry editor provides access to netlist (design) data, and a DRC tool provides the functionalities to merge polygons and to measure the distance between the edges of two layout polygons. Since every external tool is very unique in its functionality and the design data it processes, a specific TIK is required for each connected design tool.

Another extension point of the CES regards its internal handling of constraints. The CES enables the integration of arbitrary constraint solvers that are directly connected to its CLP core. The standard solver currently considers linear arithmetic constraints and nonlinear constraints that can be subsequently reduced to linear constraints. This solver is very efficient due to the use of the simplex algorithm.

In addition, the meta-verification rule developer can define new constraint solvers via CHR. The flexibility of CHR allows the definition of reusable solvers that are highly tailored to a specific constraint satisfaction problem. If neither the extended linear constraint solver nor the definition of new solvers via CHR leads to a suitable solution, new constraint solvers can be added via this extension point to the CES core. It is, for instance, expected that the resolution of polynomial and statistical constraints within CHR would not lead to constraint solvers that are efficient enough to handle complex constraint problems of that domains. Hence, specific solver could be added that resolve these constraint problems more efficiently.

Constraint compliance is the main matter of interest in a CES application. As previously mentioned, meta-verification ensures that all complex constraints are fulfilled by the design result. The definition of meta-verification rules within the CES is simple. Figure 7.3, where several simple constraints form a complex constraint, can be used as an example.

It is advisable for the demonstration to slightly modify the complex constraint C_c in Fig. 7.3, so that all star-shaped nets within an IC design can be reported whose IR-drop between two pins is greater than a maximal allowed IR-drop $V_{\text{IR-max}}$. The following CES meta-verification rule depicts the definition of such a deduction using CLP:

```
starShapedIRDrop(P1, P2, V, Virmax) :-
   starShaped(N), netPin(N, P1), netPin(N, P2),
   irDrop(P1, P2, V), V > Virmax.
```

The predicate `starShapedIRDrop(P1,P2,V,Virmax)` encapsulates C_c so that it can be reused for other verification purposes. To obtain all pins of star-shaped nets that do not meet the criterion $V_{\text{IR}} \leq 0.1\,\text{V}$, the following query is to be submitted to the CES:

```
starShapedIRDrop(P1, P2, V, 0.1).
```

With that query, the CLP core tries to find suitable bindings for the unbound variables `P1`, `P2`, and `V`. If a solution is found, the CES reports a tuple consisting of two pins and the actual IR-drop between these pins. The search can be continued until all solutions, i.e., star-shaped nets violating the IR-drop constraint, are found.

The example of the complex constraint C_c in Fig. 7.3 demonstrates the application of constraints that originate from different external sources. The simple constraint C_4 instruments an external tool that is capable of computing the IR-drop between two given pins in a net layout. From the verification point of view, C_4 is a standard relation like the other constraints of this example. The CES then forwards the calculation of the IR-drop to an external IR-drop calculation tool. The transformation of parameters and the evaluation are performed by the TIK of this tool. The same applies to all other constraints with the difference that C_1, C_2, and C_3 originate from a layout editor tool. Finally, the constraint C_5 is evaluated by the build-in arithmetic constraint solver.

Regarding the constraint sensitivity analysis example illustrated in Fig. 7.6, the sensitivity of the wire width w and the temperature T can be determined with the CLP example below. To enable the CSA, the sensitivity variables need to be limited. The temperature T in this example should range from 218 to 448 K and the wire width w from 0.18 to 2.0 µm. These ranges are added as additional constraints to the temporary constraint list.

```
{T>=218, T<=448, W>=0.18e-6, W<=2e-6}
  @ csa(V,[W,T],[SW,ST]).
```

The `csa` predicate performs the actual sensitivity analysis. The first argument denotes the target function represented by the variable `V` ($= V_{\text{IR}}$), the second a list of variables for which the sensitivity has to be determined (`W` $= w$ and `T` $= T$), and the last argument the resulting list of normalized sensitivity coefficients.

The CES provides a graphical user interface that simplifies the practical work with meta-verifications. The graphical user interface provides a uniform access to

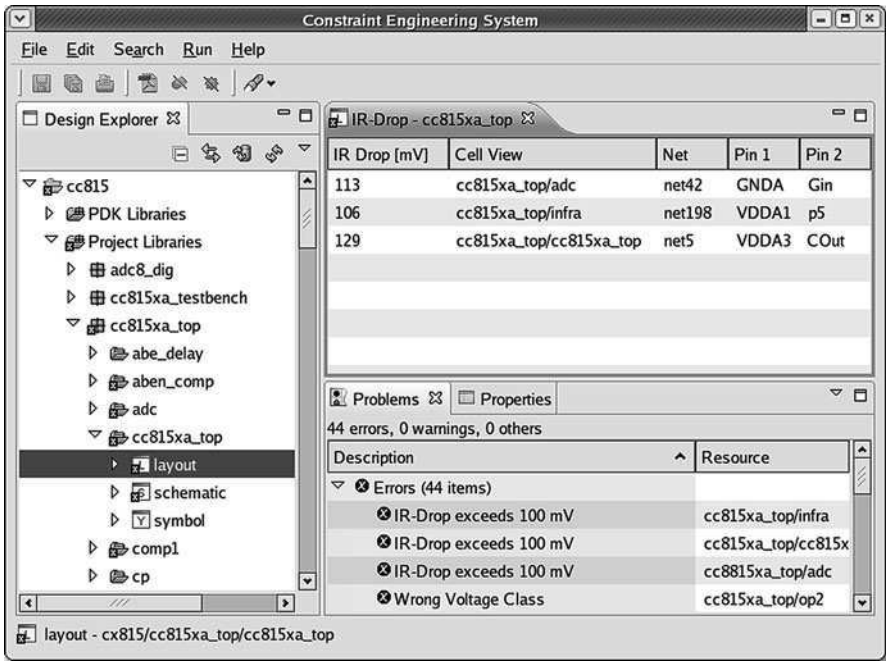


Fig. 7.9 The tabular result of the IR-drop constraint verification applied to an IC design. The shown star-shaped nets contain pin-to-pin connections having an IR-drop $V_{IR} > 0.1$ V

all meta-verification runsets that are associated with an IC design. Queries can be task-centric chosen and executed by a designer from the user interface. This releases the designer from the burden to manually specify verification queries. Figure 7.9 depicts the result of the previously described `starShapedIRDrop` query that has been applied to an IC design.

Constraint transformation, assignment, and derivation can be directly obtained by providing assignment and transformation rules using CLP and CHR. The CES regards the reuse aspect such that these rules can be applied to multiple IC designs. The CES also supports multiple process technologies by providing specific technology properties as well as an abstraction of technology properties.

7.6 Impact Analysis

In this section, we discuss the impact of an automated constraint-driven approach on the overall IC design flow, the core design of algorithms used for design automation and the required paradigm adjustments for analog physical design.

7.6.1 Impact on Design Flow

A holistic approach to analog design automation requires several new design flow components to enable an automated constraint-driven IC design. The components of the constraint-driven design flow, such as constraint management, derivation, transformation, constraint sensitivity analysis and verification have been introduced in Sect. 7.4. Hereafter, they will be denoted as "new design flow components" whose impact on the analog IC design flow will be discussed in this section.

The new design flow components complement the existing analog IC design flow. They must be perpetually available during all design stages to allow a comprehensive derivation, application and verification of constraints throughout the entire design process (Fig. 7.4). Any breach of the constraint application can lead to inconsistent design and constraint data, and hence, to a reduction of constraint verification coverage and an inconclusive verification result. The persistent use of automatic constraint verification offers greater verification coverage and reproducibility than manual verification.

All utilized design tools must fully understand the syntax and semantics of the used constraint representation. If different constraint representations exist within the design flow, then constraints must be converted between design tools that are mutually linked by a particular design task (e.g., conversion of device placement constraints within a layout editor to be used by a connected external third-party layout compaction tool). Furthermore, linked tools must support all constraint types that are relevant within a particular design context.

Constraint verification complements existing verification methods (e.g., DRC and LVS) required for sign-off in order to guarantee the intended circuit functionality. The achievable verification coverage depends on the traits and capabilities of the constraint verification framework as well as on the set of verification rules (Sect. 7.4.1). The chance of design iterations may increase if constraint verification is applied consistently due to better verification coverage. A back-annotation of constraints and verification results is required in order to minimize these iterations by addressing only relevant violations.

The constraint management system must guarantee low-level constraint data consistency by keeping each constraint and its referencing design object synchronized. Additionally, the high-level constraint data consistency, i.e., the maintenance of design data and constraint data as single data entity on file and cellview level, must be guaranteed by design guidelines and design data management systems.

7.6.2 Impact on Design Methods

Several challenges have to be addressed for a successful practical application of constraint-driven design. Among others, these challenges comprise new responsibilities for designers and the way how designers communicate with each other. The impact of these challenges is strongly dependent on the structure of the design team and the IC applications to be designed.

Several challenges arise from the change of design responsibilities since designers must now provide *all* necessary constraint information in a formalized fashion. This may lead to additional and possibly error-prone design work, whose effort must be considered in the project schedule.

As demonstrated in Fig. 7.2, the analog IC design flow exhibits overlapping design steps to account for concurrent design problems. This is partially addressed by assigning constraints and using them in subsequent design steps. Here, the key question is to clarify which constraints are to be defined at which design step. This question can be answered with good confidence for constraints having an immediate impact in the next design step. Unfortunately, it cannot be easily answered for constraints either having a continuous impact or only having an impact on remote design steps. Here, designers must currently rely on their expert knowledge while future research should address this problem.

The assignment of constraints also has an impact on the partitioning of now separated design tasks with many positive but also negative effects. While the availability of complete constraint information may now allow the use of fully constraint-driven design tools, there is also an increasing chance of over-constraining. An over-constraining done in a previous design step may aggravate or even prevent an optimization in a later design step. After performing a root cause analysis to identify that cause over-constraints designers may consider two options: (a) return to a previous design step while avoiding the causing over-constraints (design iteration), (b) override or elimination of the causing over-constraints and continuation. If root causes cannot be found then unwanted design iterations are very likely. The elegant consideration of over-constraints is a critical issue which strongly influences the acceptance and practical success of a constraint-driven design flow. This consideration is also subject to further research.

Simultaneous semi-automatic and manual design styles must complement each other as long as the relevant constraint types cannot be considered at all or in case their consideration is limited to a specific design context only. For example, in the latter case a constraint would only be considered by a design algorithm within a cellview instead of considering it within the design hierarchy (e.g., hierarchical IR-drop constraint).

To address the tight interaction between these design steps and to consider the concurrent nature of the analog design problem, all artificially introduced boundaries between existing design steps should be gradually dissolved in the future. The removal of degrees of design freedom should occur gradually rather than abruptly to keep them available for design optimization as long as possible. While the automatic approach to achieve this goal is still subject to further research, this issue is also of relevance for semi-automatic and manual design. In current analog design approaches, the strategy by which the degrees of design freedom are removed strongly depends on the designer's expert knowledge and the design task partitioning in a design team.

The reuse of analog IP often fails because small differences may prevent a direct IP reuse. A direct reuse is often not feasible if all degrees of design freedom were already removed from an IP block. However, the consistent definition of constraints

between design objects allows design reuse of structural information based on IP templates such as circuit and layout templates that already include constraints. The structural information represents the most valuable part of the design knowledge, and hence, it enables a more flexible reuse since relevant degrees of design freedom are not fixed yet. In that respect, analog design automation should address low-level layout generation and high-level design planning as discussed in the next subsection.

7.6.3 Impact on Design Algorithms

In this section, we discuss the impact constraint-driven design has on design algorithms and design planning. Furthermore, we briefly discuss new concepts and ideas for constraint-driven IC design. While some of these design approaches are new, others, such as the application of the constraint sensitivity analysis or the introduction of standardized algorithm interfaces, have already matured and thus have led to new insights into the analog design problem [14, 23, 25].

Present design algorithms are special-built for a particular purpose (e.g., focusing on placement, global or detailed routing). While this provides several benefits, such as an optimized execution time and memory footprint, it also introduces several significant limitations to "conventional design algorithms", such as incompatible interfaces for design and constraint data and a lack of functional abstraction. These limitations aggravate further advances in analog design automation.

A primary limitation in conventional algorithm design is the narrow focus on fast, but low-level execution without an implementation of standardized data interfaces. A standardized data interface creates a layer around a core algorithm to enable a common understanding of the syntax and semantic of the design and constraint data representation. This layer connects a design algorithm to the design and constraint databases as well as to other concurrently executed design algorithms. Thus, all design algorithms share a common understanding of the syntax and semantic of the design and constraint data representation.

Standardized algorithm interfaces enable the modularization and abstraction of design algorithms. The abstraction of their algorithmic work greatly improves algorithm reuse and flexibility because a single algorithm can be used to solve similar design tasks (this concept is similar to algorithm abstraction available in various programming languages). In turn, this flexibility enables the construction of high-level design algorithms that utilize modularized low-level design algorithms to perform specific design tasks on a higher level of abstraction.

The strategy in which the degrees of design freedom are removed must be carefully chosen as mentioned earlier. A removal strategy can be applied to actuate high-level design algorithms. The actuation greatly benefits from the constraint sensitivity analysis (CSA, see Sect. 7.4.4).

First, CSA can be used to identify design task parallelism by searching for temporary groups of design variables and constraints that are either not or only weakly coupled (dynamic design task partitioning). For these groups, the next design step

can then be performed independently of each other. Note that the independency of design variables and constraints in these groups may only be temporary, and hence, may not exist anymore after the design step is completed.

Second, CSA can also be used to determine the most sensitive design parameters in a particular design context that will more likely violate a constraint than non-sensitive parameters. Sensitive design parameters could then be considered with a higher priority within the specific design context.

A dynamic hierarchy of concurrent design tasks can be established in which design algorithms perform functional transformations (instead of conventional distinct design tasks) (Sect. 7.2.1). These transformations could be governed by either a fixed execution regime or more flexible approaches, such as high-level design planning algorithms that are guided by a design strategy.

Another major advantage for the development of high-level design algorithms is the possible dynamic consideration of new constraint types without the need to introduce major low-level algorithm changes. High-level design strategies can be used to solve low-level design problems by eliminating degrees of design freedom in a top-down methodology. This approach typically leads to better design results because low-level constraints are now less likely to break high-level constraints (Sects. 7.4.2 and 7.4.3).

Most of these introduced approaches promise great potential, namely the dynamic design task partitioning, the actuation of high-level design algorithms and the replacement of conventional algorithms by a sequence of continuous functional transformations. Nevertheless, all of them are still subject to further research.

7.7 Outlook

Despite the recent advances in constraint-driven design for analog IC design, there are several problems that need to be addressed in the near future to further broaden the applicability of analog design automation approaches. Methods to check the completeness of a set of constraints and constraint (meta-)verification rules, as well as the achieved verification coverage, must be developed to guarantee IC functionality, reliability, robustness, etc. The set of meta-verification rules must be optimized to allow time-efficient constraint verification. Today, such optimization is done manually but automatic rule-optimization methods should be developed to reduce this burden.

As mentioned earlier, constraint sensitivity analysis is a powerful tool to drive and support high- and low-level design decisions, and to develop high-level design algorithms that allow more gradual IC design. The scalability of existing constraint-sensitivity analysis approaches is still limited to a few thousand design variables. This is sufficient for mid-sized analog blocks with typically several hundreds of analog devices. Application to top-level design problems requires the development of new complexity reduction methods, as well as fast constraint sensitivity calculation methods to improve scalability.

Key factors for next generation analog design automation are design techniques that reduce the degree of design freedom gradually rather than abruptly while performing several conventional design steps concurrently. This will require that the current artificial boundaries between conventional design steps be (gradually) dissolved. While breaking with conventional design approaches, this paradigm change could lead to a new class of (higher level) design algorithms that brings us one step nearer to the goal of full-scale analog design automation.

Acknowledgment We would like to thank Jürgen Scheible of Robert Bosch GmbH and Ammar Nassaj of IFTE at Dresden University of Technology for the many fruitful discussions related to the topic of this chapter.

Glossary

Constraint Constraints define relations between values of design variables. Constraints defining a single relation are denoted as simple constraints. Constraints defining a set of interdependent relations are denoted as complex constraints.

Constraint Assignment Process of linking constraints to design objects. In case the corresponding design objects are located in different design hierarchy levels, the linking is done by traversing the hierarchy tree either strictly top-down, strictly bottom-up or in a mixed top-down and bottom-up manner. The link can be permanent or temporary.

Constraint Derivation Process of deriving constraints from design objectives. Constraint derivation is also known as constraint generation.

Constraint-Driven Design Design paradigm that considers all constraints in a consistent and comprehensive manner.

Constraint Engineering Design paradigm that comprises the use of several design flow components, such as constraint assignment, derivation, propagation, transformation, and verification.

Constraint Engineering System (CES) Software architecture that implements the constraint engineering concept so that all components of the constraint-driven design flow are available during the design process [11].

Constraint Handling Rules (CHR) Programming language that, among others, allows the definition of problem-specific constraint solvers [12].

Constraint Logic Programming (CLP) Form of constraint programming, in which logic programming is extended to include concepts from constraint satisfaction. The unification process in CLP is extended by constraint handling in the boolean, real or integer constraint domain [13]. CLP is often implemented as an enhancement of Prolog-like computer languages with additional constraint solving mechanisms.

Constraint Management Software architecture to enable the storage, management, access, and synchronization of constraint data. Features of the constraint management are used by all components of the constraint-driven design flow.

Constraint Programming Programming paradigm where relations between variables are stated in the form of constraints.

Constraint Satisfaction Problem (CSP) Mathematical problem defined as a set of objects whose state must satisfy a number of constraints. These problems represent the entities in a problem as a homogeneous collection of finite constraints over variables.

Constraint Sensitivity Analysis (CSA) Method to determine the sensitivity of a design parameter in relation to an objective function and related constraints.

Constraint Solver Mechanism to solve a given constraint satisfaction problem.

Constraint Transformation Process of transforming a higher level constraint into a set of lower level constraints of the same or a different domain and vice versa (inverse constraint transformation).

Constraint Type Type of a constraint that corresponds to the type of design variables which share a relation defined by that constraint.

Constraint Verification Verification process to ensure that no over-constraints exist and that all constraints are fulfilled by the design result [11].

Design Context Local context in which a particular design task is performed.

Design Object Data object represented in the database of a design tool, such as cell, cellview, instance, net, terminal, etc.

Design Objective Design goal to be achieved or specification requirement to be met by either a final or a partial design result.

Design Rule Check (DRC) Verification process to ensure that all manufacturing-related constraints are fulfilled by the design result.

Design Tool Software tool for IC design generation and verification.

Expert Knowledge Entity of a designer's problem-specific design knowledge in formalized and nonformalized form.

Layout Versus Schematic (LVS) Verification process to ensure that a given device netlist matches a netlist extracted from the layout representation.

Logic Programming (LP) Software language paradigm based on logic, more specifically on resolution theorem proving in the predicate calculus [28].

Meta-Verification Verification process to ensure that all complex constraints are fulfilled by the design result [11].

Over-Constraint Condition in which not all given constraints can be fulfilled simultaneously.

Predicate (→ LP, CLP) Mathematical sentence that describes a common property by which a subset of objects can be identified within a global set of objects.

Propagation (→ CHR) The propagation within CHR is the derivation of one or more new constraints from a given set of constraints. It is triggered by the existence of one or more constraints that are already part of the constraint set. After the propagation took place, the new constraints are part of the constraint set [12].

Root Cause Analysis Class of problem solving methods aimed at identifying the root causes of problems or events. One approach of solving an existing design problem is to eliminate its root causes. Root cause analysis is often used iteratively (continuous improvement).

Schematic-Driven Layout (SDL) Design paradigm in which the layout generation is driven by the schematic representation of the circuit.

Simpagation (→ CHR) The simpagation is a combined application of propagation and simplification. While it can be expressed by solely using propagation and simplification rules, it can be handled more efficiently [12].

Simplification (→ CHR) The simplification within CHR removes one or more constraints from a given set of constraints. It is triggered by the existence of one or more constraints that are already part of the constraint set [12].

Tool Integration Kit (TIK) Data interface of the constraint engineering system that translates tool-specific data into the CLP language and vice versa.

Unification (→ CLP) Process that tries to match symbolic expressions by assigning subexpressions to variables that are part of two expressions [2]. Unification is a core concept of logic programming.

References

1. S. Abdennadher, E. Krämer, M. Saft, and M. Schmauss. JACK: A Java constraint kit. In *Proc. Int. Workshop Functional and (Constraint) Logic Programming*, volume 64, pages 1–17. Elsevier B.V., Amsterdam, 2001.
2. F. Baader and W. Snyder. *Handbook of Automated Reasoning*, volume 1, Unification Theory, pages 445–533. Elsevier Science B.V., Amsterdam, 2001.
3. R. Barták. Theory and practice of constraint propagation. In *Proc. 3rd Workshop Constraint Programming for Decision and Control (CPDC)*, pages 7–14, 2001.
4. D. G. Cacuci, M. Ionescu-Bujor, and I. M. Navon. *Sensitivity & Uncertainty Analysis: Applications to Large-Scale Systems*, volume 2. Chapman & Hall/CRC, 2005.
5. Cadence Design Systems, Inc. http://www.cadence.com.
6. J. A. Carballo and S. W. Director. Constraint management for collaborative electronic design. In *Proc. IEEE/ACM 36th Design Automation Conference (DAC)*, pages 529–534, Berlin, 1999.

7. H. Chang, E. Charbon, U. Choudhury, A. Demir, E. Felt, E. Liu, E. Malavasi, A. Sangiovanni-Vincentelli, and I. Vassiliou. *A Top-Down, Constraint-Driven Design Methodology for Analog Integrated Circuits*. Springer, 1999.
8. J. Cohen. Constraint logic programming languages. *Commun. ACM*, 33(7):52–68, 1990.
9. M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proc. Int. Conf. 5th Generation Computer Systems*, pages 693–702, 1988.
10. B. M. Freeman-Benson. *Constraint Imperative Programming*. PhD thesis, University of Washington, Department of Computer Science and Engineering, 1991.
11. J. Freuer, G. Jerke, J. Gerlach, and W. Nebel. On the verification of high-order constraint compliance in IC design. In *Proc. IEEE/ACM Int. Conf. Design Automation and Test in Europe (DATE)*, pages 26–31, 2008.
12. Th. Frühwirth. Introducing simplification rules. Technical Report ECRC-LP-63, European Computer-Industry Research Centre, Munich, Germany, 1991.
13. Th. Frühwirth, A. Herold, V. Küchenhoff, Th. Le Provost, P. Lim, E. Monfroy, and M. Wallace. Constraint logic programming – an informal introduction. *Lecture Notes In Computer Science*, 636:3–35, 1992.
14. G. J. Gad El-Karim, R. S. Gyurcsik, and G. L. Bilbro. Sensitivity-driven placement of analog modules. In *Proc. IEEE Int. Symp. Circuits and Systems (ISCAS)*, pages 363–366, 1994.
15. V. Gerard and Th. Schiex. Solution reuse in dynamic constraint satisfaction problems. *Proc. Association for the Advancement of Artificial Intelligence (AAAI)*, pages 307–312, 1994.
16. H. Gräb, F. Balasa, R. Castro-Lopez, Y.-W. Chang, F. V. Fernandez, P. -H. Lin, and M. Strasser. Analog layout synthesis – recent advances in topological approaches. In *Proc. IEEE Int. Conf. Design Automation and Test in Europe (DATE)*, pages 274–279, 2009.
17. M. Grabmüller and P. Hofstedt. Turtle: A constraint imperative programming language. In *Proc. 23rd SGAI Int. Conf. Innovative Techniques and Applications of Artificial Intelligence*, 2003.
18. D. R. Insua. *Sensitivity Analysis in Multi-Objective Decision Making*. Springer, Berlin, 1990.
19. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\Re$) language and system. *ACM Trans. Programming Languages and Systems*, 14(3):339–395, July 1992.
20. V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1): 32–44, 1992.
21. E. Malavasi and E. Charbon. Constraint transformation for IC physical design. *IEEE Trans. Semiconductor Manufacturing*, 12(4):386–395, 1999.
22. E. Malavasi, E. Charbon, B. Arsintescu, and W. Kao. A constraint management system for IC physical design. In *Proc. 11th Brazilian Symp. Integr. Circuit Design*, pages 240–243, 1998.
23. E. Malavasi, E. Charbon, E. Felt, and A. Sangiovanni-Vincentelli. Automation of IC layout with analog constraints. *IEEE Trans. CAD of Integr. Circuits and Systems*, 15(8):923–941, 1996.
24. Mentor Graphics Inc. http://www.mentor.com.
25. P. Miliozzi, I. Vassiliou, E. Charbon, E. Malavasi, and A. Sangiovanni-Vincentelli. Use of sensitivities and generalized substrate models in mixed-signal IC design. In *Proc. IEEE/ACM 33rd Design Automation Conference (DAC)*, pages 227–232, 1996.
26. A. Nassaj, J. Lienig, and G. Jerke. A new methodology for constraint-driven layout design of analog circuits. In *Proc. IEEE Int. Conf. Electronic, Circuits and Systems (ICECS)*, pages 996–999, 2009.
27. J.-F. Puget. A C++ implementation of CLP. Tech. rep. 94-01, ILOG SA, Gentilly Cedex, France, 1994.
28. J. A. Robinson. A machine-oriented logic based on the resolution principle. *ACM*, 12(1): 23–41, 1965.
29. R. A. Rutenbar and J. M. Cohn. Layout tools for analog ICs and mixed-signal SoCs: A survey. In *Proc. IEEE/ACM Int. Symp. Physical Design (ISPD)*, pages 76–83, 2000.

30. J. Scheible. Constraint-driven Design – Eine Wegskizze zum Designflow der nächsten Generation (in German). In *Proc. VDE ANALOG'08*, 2008.
31. G. Tack. *Constraint Propagation – Models, Techniques, Implementation*. PhD thesis, Saarland University, 2009.
32. M. Wallace, S. Novello, and J. Schimpf. ECL^{i}PS^e: A platform for constraint logic programming. Technical report, IC-Parc, Imperial College, London, UK, 1997.

Index

A
A* algorithm, 174, 176, 191
A–G parasitic extraction, 262
Absolute representation, 8
Analog arrays, 192
Analog layout generation, 244
Analog sizing, 244
Analytical parasitic extraction, 262
Area loss, 254, 257, 259
Area routing, *155*, 169, 172, 176, 191
Area usage, 133
ASF-B*-tree, *67*, 70–81, 83, 88, 91, 92
Automatically symmetric-feasible B*-tree, *see* ASF-B*-tree

B
B*-tree, 63, *66*, 67, 69, 71, 73, 74, 76, 82, 83, 88, 92, 115
Backtracking, 151, 182, 183
Basic enumeration, 129
Boundary constraint, *see* Constraints boundary
Bounded-sliceline grid (BSG), 10
Bounding constraints, 172, 174
Bridging fault, 195

C
Cadence's PCELLS, 252
Cadence's SKILL language, 252
Catalan numbers, 50
Center of gravity, 96
  groups, 97
CES, *see* Constraint Engineering System
Channel
  intersection graph, 157, 182
  routing, 152, 166, 172
CHR, *see* Constraint Handling Rules
Clause, 275
CLP, *see* Constraint Logic Programming
COG, *see* center of gravity
Combinatorial optimization, 6
Congestion, 167, 170, 171, 174, 180, 181, 190, 191
Connectivity, 159, 180, 185, 192
Constraint graph, 153, 172, 185, 209
  basic coefficient, 219
  coefficient, 217
  gain equation, 221
  horizontal, 115, 153
  long wires, 211
  longest-path algorithm, 211
  loops, 225
  mathematics, 222
  multi-variable simplex, 217, 221
  simplex, 216
  super-node, 218
  vertical, 115, 153
Constraints, 293
  assignment, *277*, 288, 293
  automatic generation, 104–114
  boundary, 69
  category, 273
  classification, 273
  common centroid, 61, 62, 98
  complex constraint, *273*, 274, 275, 278, 279, 282, 286, 287, *293*
  derivation, 275, 278, 283, 284, 288, 289, *293*
  distance, 210
  driven design, 276, 286, 288, *291*, *293*
  engineering, 271, 284, *284*, *293*
  engineering system, 285, 286, *286*, 288, 293
  extraction, 186
  generation, 171, 174, 209
  graph, *see* Constraint graph
  handling rules, 285, 293

importance order, 105
layout, 231
logic programming, 275, 282, 284, *293*
management, *275*, 276, 289, 294
matching, *104*, 108, 172–174
meta-verification, *282*, 286, 287, 294
minimum distance, 99, 117
multi-variable, 212
over-constraint, *276*, 281, 282, 290, 295
piecewise-linear minimum distance, 100, 120
programming, 284, 294
propagation, 277, 285, 295
proximity, 98, 107
representation, 273
satisfaction problem, 276, 283, 294
sensitivity analysis, 275, *279*, 281, 289, 291, 292, 294
simple constraint, 273–275, 279, 287, *293*
solver, *284*, 286, 287, 294
symmetry, 61, *63*, 77, 82, 83, 86, 91, 98, 108
transformation, 275, 278, *278*, 288, 289, 294
type, 272, 274, 284, 289, *294*
uniform representation, 269, *275*
variant, 98
verification, 276, 281, 283, 288, 289, 294
Contour node, 73–77, 80, 82
Corner block list (CBL), 10
Critical net, 167, 168
Crosstalk, 161, 164, 171, 175, 179, 185, 191
CSA, *see* Constraint Sensitivity Analysis
CSP, *see* Constraint Satisfaction Problem

D

Deep trench isolation, 100, 120
Design
algorithm, 272–275, 279, 280, 283, 286, 290, 291
automation, 270, 272, 278, 284, 293
context, 272, 278, 289, 294
freedom, 271, 279, 290, 292
gap, 272
hierarchy, 274, 277, 290
methodology, 270, 289, 292
object, 289, 294
objective, 269, 272, 294
problem, 271, 275, 292
task partitioning, 290–292
tool, 272, 286, 289, *294*
Design Rule Check, 273, 283, 289
Design rule check, *294*
Design rules, 158, 161, 168, 179, 184–186
Detailed routing, *166*, 167, 168, 178, 180, 183, 195
Deterministic optimization techniques, 247
Deterministic skip list (DSL), 25
Device, 96
recognition, 208
replacement, 236
Device merging, 5
Distance between modules, 96
DRC, *see* Design Rule Check
DTI, *see* deep trench isolation

E

Electrical rules, 161
Electrical synthesis, 244
Electromigration, 195
EM simulations, 165
Enhanced shape functions, 122
combination, 124, 130
Enhanced shapes
horizontal addition, 125
vertical addition, 125
Estimation, 153, 167, 169, 181, 182
Expert Knowledge, 269, 278, 290, *294*
Extraction, 159, 161, 164

F

Flat representation, 8
Floorplan sizing, 254, 256
Floorplan-sizing matrix, 257
Functional Transformation, 271, 273, 292

G

Genetic algorithms, 6
Geometric Constraints Module, 253
Geometric constraints module, 256, 262
Geometric parasitic extraction, 262
Geometrically constrained sizing, 245, 253
Geometry sharing, 5
Global routing, 155, *166*, 167, 168, 178, 180, 182, 190, 191
Grid, 151, *155*, 158, 168–170, 174, 179, 190
graph, 156
Group
basic, 96
hierarchical, 97
matching, 111
of modules, 96
proximity, 111
symmetry, 111

H
HB*-tree, *see* Hierarchical B*-tree
HCG, *see* constraint graph, horizontal
Hierarchical B*-tree, 63, *72*, 75–77, 80, 81, 83–88, 91, 92
Hierarchical proximity, 62, 63
Hierarchical symmetry, 62, 63, 91
Hierarchy, 161, 168, 184
Hierarchy node, 72–77, 81, 88, 92
HSMPG tree, 111, 128

I
Integrated placement and routing, 178, 179

J
Johnson's priority queue, 33

K
Knowledge-based layout synthesis, 247
Knowledge-based sizing, 246

L
Layout
  compaction, 209, 212, 216
  description script, 185
  design-rule, 208, 209, 232, 233
  hierarchy, 236
  language, 183, 184
  layer mapping, 208
  migration, 207
  parasitics, 237
  retargeting, 207, 234
  symmetry, 234
Layout aware's geometric goals, 259
Layout design hierarchy, 62, 63, 92
Layout geometric parameters, 254, 262
Layout parasitics, 244, 262
Layout sampling, 263
Layout slicing style, 254
Layout Versus Schematic, *273*, 283, 289, 294
Layout-aware sizing, 245
Layout-aware's geometric goals, 258
Line expansion, 152, *154*, 158, 163, 169, 171, 194
Linear Programming, 212
Logic Programming, 294
Longest common subsequence (LCS), 37
LP, *see* Logic Programming
LVS, *see* Layout Versus Schematic

M
Manufacturability, 194
Matching constraints, *see* Constraints matching
Matching requirement, 108
Maze router, *151*, 154, 167, 174, 182
Meta-Verification, *see* Constraint Meta-Verification
Mirror symmetry, 7
Module, 96

N
Net ordering, *154*, 167, 168, 170, 190
Net splitting, 160, 162
Non-slicing placement, 10
Numerical parasitic extraction, 262

O
O-trees, *see* Ordered trees
Optimization-based layout synthesis, 247
Optimization-based synthesis, 247
Ordered trees, 10

P
Parasitic extraction techniques, 262
Parasitic-aware sizing, 245
Parasitics estimates, 245
Path search, 158, 165, 176
Perfect symmetry, 8
Performance-driven analog placement, 6
Perturbation method, 172
Physical synthesis, 244
Powell's method, 251
Predicate, 287, 295
Probabilistic routing, 190
Probabilistic skip list, 25
Proximity constraint, 61
Proximity requirement, 107
PSL, *see* probabilistic skip list

R
Red-black interval tree, 18
Red-black tree, 18
Representative B*-tree, 67–70
Retargeting, 184
Reuse-based design, 245
RF circuits, 165, 191
Rip-up and re-route, *159*, 167, 170, 171, 174
Root Cause Analysis, 290, 295
Rooted binary trees, 254

Routing
analog circuits, 149, 151, 153, 155, 157, 159, 161, 163, 165, 167, 169, 171, 173, 175, 177, 179, 181, 183, 185, 187, 189, 191, 193, 195, 197, 199, 201
parasitics, 162, 167, 170, 172, 191
Rule-based layout synthesis, 247

S
Schematic-Driven Layout, 270, 295
SDL, *see* Schematic-Driven Layout
Segment tree, 12
Self-symmetry, 8
Sensitivity, 169, 171, 173–175, 180–182
Sequence Pair, 63, 82–87
Sequence-pair, 10
Shape function, 256
Shape functions, 122
Sheet resistance, 162
Shield, *164*, 167, 174, 175
Simpagation, 285, 295
Simplex, 212
basic feasible solution, 213, 214, 226
entering variable, 215
graph-based, 216, 217, 221, 237
leaving variable, 215
revised, 234, 237
steps, 214
Simplification, 285, *295*
Simulated annealing, 6, 63
Simulated annealing optimization, 250
Simultaneous placement and routing, 158, 179, 180
Slicing floorplan, 254
Slicing placement, 9
Slicing tree, 10
Stasheff polytope, 52
Statistical optimization techniques, 247
Steiner tree, *154*, 168, 180, 181, 193, 195
Stockmeyer's algorithm, 256
Subnet, 160
Symmetric
net, 175
routing, 162, 182
wiring, 170
Symmetric-feasible (S-F) binary tree, 51
Symmetric-feasible (S-F) sequence-pair, 35
Symmetry constraint, *see* Constraints symmetry
Symmetry group, 8
Symmetry island, 65, *65*, 66, 67, 70–76, 80–82, 84, 88, 91, 92
Symmetry requirement, 108
Symmetry-island, 63

T
Table lookup parasitic extraction, 262
Template-based approaches, 183
Template-based layout synthesis, 247, 251
Template-driven layout generation, 6
TIK, *see* Tool Integration Kit
Tile, 156, 158, 180
Tool Integration Kit, 285, 286, *295*
Topological, 158, 162
Topological representation, 9
Transitive closure graph (TCG), 10
Traversal of a binary tree
inorder, 51, 125, 127
postorder, 51
preorder, 51, 125, 127
Two-step routing, *166*, 168, 178, 182

U
Unification, 275, *295*

V
Variants, 97
VCG, *see* constraint graph, vertical

X
X-y routing, 190

Y
Yield, 194